Query Execution on Modern CPUs

Dissertation
zur Erlangung des akademischen Grades
doctor rerum naturalium (Dr. rer. nat.)
im Fach Informatik

eingereicht an der

Mathematisch-Naturwissenschaftlichen Fakultät
der Humboldt-Universität zu Berlin
von
M.Sc. Steffen Zeuch

Präsidentin der Humboldt-Universität zu Berlin:
Prof. Dr.-Ing. Dr. Sabine Kunst
Dekan der Mathematisch-Naturwissenschaftlichen Fakultät:
Prof. Dr. Elmar Kulke

Gutachter:
1. Prof. Johann-Christoph Freytag, Ph. D.
2. Prof. Dr.-Ing. Wolfgang Lehner
3. Prof. Dr. Stefan Manegold

Tag der Verteidigung 27.04.2018

Abstract

Over the last decades, database system have been migrated from disk to memory architectures such as RAM, Flash, or NVRAM. Research has shown that this migration fundamentally shifts the performance bottleneck upwards in the memory hierarchy. Whereas disk-based database systems were largely dominated by disk bandwidth and latency, in-memory database systems mainly depend on the efficiency of faster memory components, e. g., RAM, caches, and registers.

With respect to hardware, the clock speed per core reached a plateau due to physical limitations. This limit caused hardware architects to devote an increasing number of available on-chip transistors to more processors and larger caches. However, memory access latency improved much slower than memory bandwidth. Nowadays, CPUs process data much faster than transferring data from main memory into caches. This trend creates the so-called *Memory Wall* which is the main challenge for modern main memory database systems.

To encounter these challenges and enable the full potential of the available processing power of modern CPUs for database systems, this thesis proposes four approaches to reduce the impact of the Memory Wall. First, SIMD instructions increase the cache line utilization and decrease the number of executed instructions if they operate on an appropriate data layout. Thus, we adapt tree structures for processing with SIMD instructions to reduce demands on the memory bus and processing units are decreased. Second, by modeling and executing queries following a unified model, we are able to achieve high resource utilization. Therefore, we propose a unified model that enables us to utilize knowledge about the query plan and the underlying hardware to optimize query execution. Third, we need a fundamental knowledge about the individual database operators and their behavior and requirements to optimally distribute the resources among available computing units. We conduct an in-depth analysis of different workloads using performance counters create these insights.

Fourth, we propose a non-invasive progressive optimization approach based on in-depth knowledge of individual operators that is able to optimize query execution during run-time. In sum, using additional run-time statistics gathered by performance counters, a unified model, and SIMD instructions, this thesis improves query execution on modern CPUs.

Zusammenfassung

Über die letzten Jahrzehnte haben sich Datenbanken von festplattenbasierten zu hauptspeicher-basierten Datenbanksystemen entwickelt. Dabei hat die Forschung in diesem Bereich gezeigt, dass der Flaschenhals sich innerhalb der Speicherhierarchie in Richtung der Schnelleren aber Langsameren Speicher verschoben hat. Wohingegen festplattenbasierte Datenbanken vor allem durch die Bandbreite und Latenzen der Festplatten limitiert waren, sind heutige hauptspeicher-basierte Datenbanken eher durch schnellere Speicher wie Hauptspeicher, Caches oder Register limitiert.

Innerhalb der gleichen Zeitpanne hat sich die Hardware heutiger Computer ebenfalls weiterentwickelt. Im Besonderen hat die Taktfrequenz der Prozessoren ein Plateau erreicht da die Prozessorhersteller an physische Grenzen gestoßen sind. Als Ausweg haben die Hersteller die zu Verfügung stehenden neuen Transistoren genutzt um zusätzliche Prozessorkerne sowie größere Caches auf den Prozessoren zu integrieren. Als zweite Entwicklung hat sich die Hauptspeicherlatenz wesentlich langsamer verbessert als die Hauptspeicherkapazität. Im Ergebnis können heutige Prozessoren Daten wesentlich schneller verarbeiten als das Ihnen Daten zugeführt werden können. Diese Entwicklung führt zur sogenannten „Memory Wall“ die eine wesentliche Herausforderung für moderne Datenbanksysteme darstellt.

Um diese Herausforderungen anzugehen und das volle Potenzial moderner Prozessoren zu erschließen, stellt diese Dissertation vier Ansätze vor um den Einfluss der „Memory Wall“ zu reduzieren.

Der erste Ansatz zeigt auf, wie spezielle Prozessorinstruktionen (sogenannte SIMD Instruktionen) die Ausnutzung von Caches erhöhen und gleichzeitig die Anzahl der Instruktionen verringern. In dieser Arbeit werden dazu vorhandene Baumstrukturen so angepasst, das diese SIMD Instruktionen verwenden können und daher die benötigte Hauptspeicherbandbreite verringert wird.

Der zweite Ansatz dieser Arbeit führt ein Model ein, dass es ermöglicht, die Anfrageausführung in verschiedenen Datenbanksystemen zu vereinheitlichen und dadurch vergleichbar zu machen. Durch diese Vereinheitlichung wird es ermöglicht, die Hardwareausnutzung durch Hinzunahme von Wissen über die auszuführende Hardware zu optimieren.

Der dritte Ansatz analysiert verschiedene Datenbankoperatoren bezüglich ihres Verhaltens auf verschiedenen Hardwareumgebungen. Diese Analyse ermöglicht es, Datenbankoperatoren besser zu verstehen und Kostenmodell für ihr Verhalten zu erstellen.

Der vierte Ansatz dieser Arbeit baut auf der Analyse der Operatoren auf und führt einen progressiven Optimierungsalgorithmus ein der die Ausführung von Anfragen zur Laufzeit auf die jeweiligen Bedingungen wie z.B. Daten- oder Hardwareeigenschaften anpasst. Dazu werden zur Laufzeit prozessorinterne Zähler verwendet die das Verhalten des Operators auf der jeweiligen Hardware widerspiegeln.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the last decades, two predominant workloads for database systems emerged. As a consequence, database vendors optimized their database systems to one of these workloads. On the one hand, database vendors optimized their database systems for *Online Transaction Processing* (OLTP) workloads. OLTP workloads consist of simple update/insert/delete queries on small operational data sets. On the other hand, database vendors optimized their systems for *Online Analytical Processing* (OLAP) workloads. OLAP workloads consist of complex reporting queries on large historical data sets. Both workloads have their unique distributions on read and write operation as well as unique requirements for transactional consistency and performance. Although some database systems execute OLAP and OLTP workloads in the same database system, the best performance is achieved by database systems which are optimized for the applied workload. This thesis focuses on OLAP database system with their high requirements on complex operations on large data sets.

Contemporary to the emergence of different database workloads, the hardware characteristics have changed significantly. Early *disk-based* database systems exploit slow disks as their main storage medium because main memory was too small and too expensive to store the entire working data set. As a consequence, data transfer from disk was the bottleneck for query execution. Thus, the time interval from issuing a data load until it arrives in the CPU was rather high, i. e., milliseconds. Over time, the capacity of main memory has increased as well as their price has decreased. This trend leads to the second generations of so-called *in-memory* database systems. In-memory database systems exploit main memory as their primary storage medium which exhibits significantly different access characteristic in terms of latency and bandwidth compared to disk. As a consequence, the bottleneck shifts to main memory where it hits the Memory Wall, i. e., the growing disparity between CPU speed and memory access latency.

Since the year 2000, the clock speed per core reached a plateau due to physical limitations. Since then, an increasing number of available on-chip transistors are used to incorporate more processors and larger cache into CPUs. Together with main memory, caches establish the so-called *memory hierarchy* where each level trades size for lookup speed. These trend shifts the bottleneck further up in the memory hierarchy such that the working set might be small enough to be stored in a certain cache entirely. However, research in the area of OLAP workloads showed [Bea13, MBK02] that the actual processing operations are seldom the performance limiting factor. The majority of time is spent for waiting on data.

In this thesis, we will contribute in the research area of OLAP workloads on modern CPUs. We will provide approaches to alleviate the main memory bottleneck and improve the overall performance and robustness of query execution on modern CPUs. By revealing the most important characteristics of modern CPUs, we will provide the foundation to increase the efficiency of database systems. Our goal is to enable the research community to exploit the tremendous capabilities of modern processors more efficiently.

1.1 Research Contributions

The primary contributions of this thesis are:

- We propose two tree adaptations which exploit SIMD instructions to speedup the tree traversal process. Therefore, we present different transformation and search algorithms to enable SIMD for tree processing. Furthermore, we evaluate our tree adaptation on different workloads and highlight their strength and weaknesses. This work was published in [ZFH14].

- We propose an unified model to describe the parallel execution of queries. Therefore, we combine knowledge about the query plan and the underlying hardware to optimize query execution on modern CPUs. Using our model, we classify common database systems and compare their query processing strategies. As a result, we are able to reason about the performance of different database systems and their executions strategies on an abstract level. This work was published in [ZF14].

- We measure and analyze different workflows on modern CPUs using performance counters. In particular, we perform an in-depth analysis of the relational selection operator. The obtained knowledge might be valuable for query optimizer to speedup query execution. This work was published in [ZF15].

- Based on the results of the in-depth analysis of database operators, we propose a non-invasive progressive optimization approach. This approach progressively optimizes query execution during run-time on modern CPUs. Our approach bases its optimization on performance counters and supports query optimizer to converge to the best query plan during run-time. This work was published in [ZPF16].

1.2 Organization of this Thesis

This thesis is structured as follows. Chapter 2 provides a general overview of modern CPUs and database systems. It introduces features of modern CPUs which are exploited in this thesis. Furthermore, it shows the current state of database research on modern CPUs. Especially, we investigate approaches which take processor characteristics into account.

In Chapter 3, we focus on vectorized execution in database system. In particular, we adapt tree structures for processing with SIMD instructions. By restructuring the layout of trees, we enable efficient SIMD operations on trees and thus speedup tree traversal significantly.

Chapter 4 investigates how the cache hierarchy in modern CPUs impacts query execution. We introduce a model for parallel query execution that highlights the differences between common database systems. This model enables us to reason about the efficiency of query execution regarding the cache hierarchy. As a result, we identify the most efficient query execution plans and explain their advantages using performance counters.

In Chapter 5, we perform an extensive case study of the relational selection operation using performance counters. We investigate different aspects such as instruction-related and data-related processor characteristics. Furthermore, we examine different components such as caches or branch prediction to create a fundamental knowledge of executing a relational selection on a modern processors.

Based on the insights of Chapter 5, we propose a progressive optimization algorithm in Chapter 6. This algorithm bases its optimization decisions on the insights of our case study and re-optimizes a query execution plan during run-time. We provide the necessary cost models as well as an efficient re-optimization algorithm that contributes only a small run-time overhead. As a result, we are able to progressively approximate the *best* query plan during run-time.

This thesis concludes in Chapter 7 with a summary of its contributions and how these might be used to further improve query execution in the future.

Chapter 2

Modern CPUs

This chapter describes the architecture of modern CPUs. With *modern* we refer to processors that exhibit the following four properties:

1. Modern CPU features to accelerate execution, i. e., pipelining, superscalar execution, out-of-order execution, and branch prediction (see Section 2.1)
2. Multi-core or simultaneous multi-threading (see Section 2.2)
3. SIMD vector instructions (see Section 2.3)
4. Memory hierarchy including caches (see Section 2.4)

In the following sections, we describe these properties of modern CPUs in detail.

2.1 Modern CPU Features

In this section, we introduce modern CPU features that accelerate the execution of programs. These features are implemented and maintained solely by the CPU hardware. Thus, they are transparent for the executed program. However, a program might be optimized to fully exploit the available capacities of modern CPUs. This section presents the hardware features that are exploited in this thesis, i. e., pipelining, superscalar execution, out-of-order execution, and the memory hierarchy.

2.1.1 Pipelining

Modern CPUs execute instructions in a time-sliced fashion inside a so-called *pipeline*. To enable a pipelined execution, each instruction is split into a fixed number of micro operations. In Figure 2.1, we plot a CPU pipeline consisting of four stages. In the first stage, the instruction is fetched from

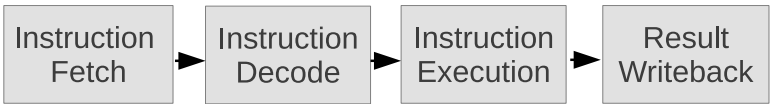


Figure 2.1: Instruction Pipeline Stages.

main memory or cache. In the second stage, the instruction is decoded into micro operations. These micro operations describe basic operations such as arithmetic or logical operations on registers, or data transfer operations from or into registers. During the third stage, the micro operation is executed by the appropriate CPU component, e. g., an arithmetic logic unit (ALU) or a memory load unit. Finally, in the fourth stage, the result of the instruction is written back into memory or into a cache.

Modern CPUs execute multiple instructions in parallel as long as their micro operations are at different stages. In Figure 2.2, we show a CPU executing three instructions in parallel. As shown, each instruction is at a different execution stage at each clock cycle. This type of parallelism is called *instruction level parallelism* (ILP). As a result, ILP speeds up the execution without changing the clock speed of the CPU. Additionally, the result of an instruction can be directly forwarded to the next instruction instead of writing it back to memory and loading it again.

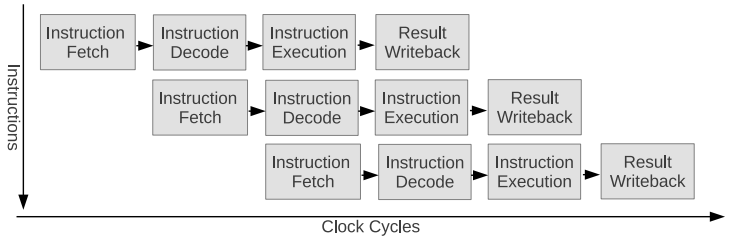


Figure 2.2: CPU Pipeline.

It is important to note, that ILP requires that all stages induce the same execution time to run efficiently. However, branches and long lasting memory accesses might block pipelines. In such a case, all subsequent instructions are stalled.

Because clock speed is bound by physical limitations such as transmission delays and heat build-up, vendors induced CPUs with deeper pipelines and a larger number of shorter stages. Using a larger number of shorter stages, the entire CPU may run at higher clock speed. As a result, each instruction will take more cycles to complete (so-called latency) but the overall throughput in terms of instructions completed per cycle is improved. In Table 2.1, we show common CPUs and their pipeline depth. Today's CPUs execute between 12 and 25 stages [Int12b, AMD13, Pat15], which provides the best performance/efficiency ratio in terms of computational capabilities and energy consumption.

Pipeline Depth	Processor
6	UltraSPARC T1
7	PowerPC G4e
8	UltraSPARC T2/T3, Cortex-A9
10	Athlon, Scorpion
11	Krait
12	Pentium Pro/II/III, Athlon 64/Phenom, Apple A6
13	Denver
14	UltraSPARC III/IV, Core 2, Apple A7/A8
14/19	Core i*2/i*3 Sandy/Ivy Bridge, Core i*4/i*5 Haswell/Broadwell
15	Cortex-A15/A57
16	PowerPC G5, Core i*1 Nehalem
18	Bulldozer/Piledriver, Steamroller
20	Pentium 4
31	Pentium 4E Prescott

Table 2.1: Pipeline Depth (Following [Pat15]).

2.1.2 Superscalar Execution

By duplicating functional units such as ALUs or memory load units, modern CPUs support the execution of multiple instructions in parallel within one cycle. This technique is called *superscalar*. In Figure 2.3, we show a superscalar execution with multiple instructions in the execution stage at the same clock cycle. This execution is possible if either two instructions require different functional units or the required functional unit is available multiple times. Using superscalar execution, the number of instructions completing every cycle (IPC) might be increased significantly.

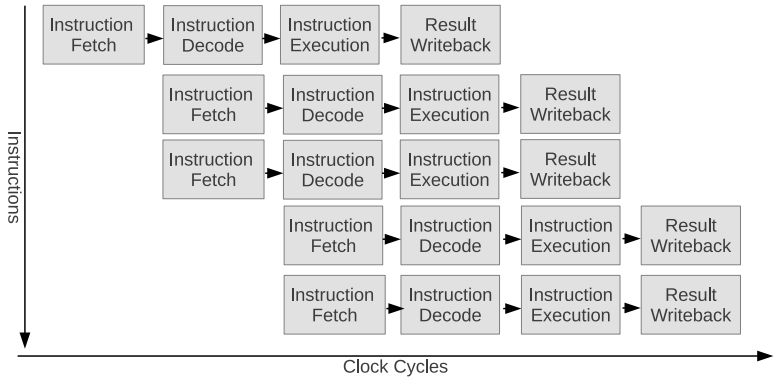


Figure 2.3: Superscalar CPU.

The number of instructions that might be issued, executed, or completed per cycle is called *processor width*. The actual width of a CPU highly depends on the executed program and its instruction mix. Therefore, different code sequences have different mixes of instructions. For example, three instructions per cycle can either be three integer instructions or one integer, one floating point, and one memory load instruction.

In Table 2.2, we summarize different CPUs and their processor width. The number and type of functional units depend on the processor and its design. Therefore, some processors have more floating point units, e. g., IBMs Power line for scientific computation, and other CPUs have more integer units, e. g., Pentium processes. Finally, the PowerPC provides more SIMD vector units. Overall, the CPU vendors try to balance the functional units for general purpose computing.

Processor Width	**Processor**
1	UltraSPARC T1
2	UltraSPARC T2/T3, Scorpion, Cortex-A9
3	Pentium Pro/II/III/M, Pentium 4, Krait, Apple A6, Cortex -A15/A57
4	UltraSPARC III/IV, PowerPC G4e
4/8	Bulldozer/Piledriver
5	PowerPC G5
6	Athlon, Athlon 64/Phenom, Core 2, Core i*1 Nehalem, Core i*2/i*3 Sandy/Ivy Bridge, Apple A7/A8
7	Denver
8	Core i*4/i*5 Haswell/Broadwell, Steamroller

Table 2.2: Processor Width (Following [Pat15]).

2.1.3 Out-of-Order Execution

The main disadvantage of the pipeline execution model is the possibility of pipeline stalls. For instance, a stall occurs if an instruction induces a long lasting memory load or a required functional unit is occupied. To mitigate the effect of stalls, instructions can be either statically or dynamically reordered. Static reordering is performed during compile-time by rearranging instructions. However, to deduce all dependencies between instructions is a complex task for any compiler. Therefore, modern CPUs contain an *out-of-order* (OoO) unit. OoO execution induces dynamic instruction scheduling (reordering) during run-time. In combination with dynamic register renaming, OoO execution provides a high degree of flexibility in terms of instruction scheduling. On the other hand, OoO execution makes CPUs less energy-efficient and more complex compared to an in-order execution of instructions.

Early processors implemented in-order instruction execution, e. g., SuperSPARC, hyperSPARC, UltraSPARC, Alpha 21064 and 21164, and the original Pentium. In contrast, early OoO CPUs are the MIPS R10000, Alpha 21264 and to some extent the entire POWER/PowerPC line [Pat15]. Except the UltraSPARC processors from Sun, all modern high performance CPUs nowadays use the OoO design [HP11, JW89, WS94]. However, in the area of low-power/low-performance computing, especially in mobile devices, CPUs like the Cortex or Atom processor use an in-order design to save power.

2.2 Multi-Core and SMT

Over the last decades, the clock speed per core reached a plateau due to physical limitations. As a result, an increasing number of available on-chip transistors are used to incorporate more processors per socket. Figure 2.4 summarizes the evolution of CPUs over the last decades. As shown, the number of transistor per core constantly increases. In contrast, the power consumption as well as the frequency per core stagnate since 2000. On the other hand, the number of cores has significantly increased since 2000.

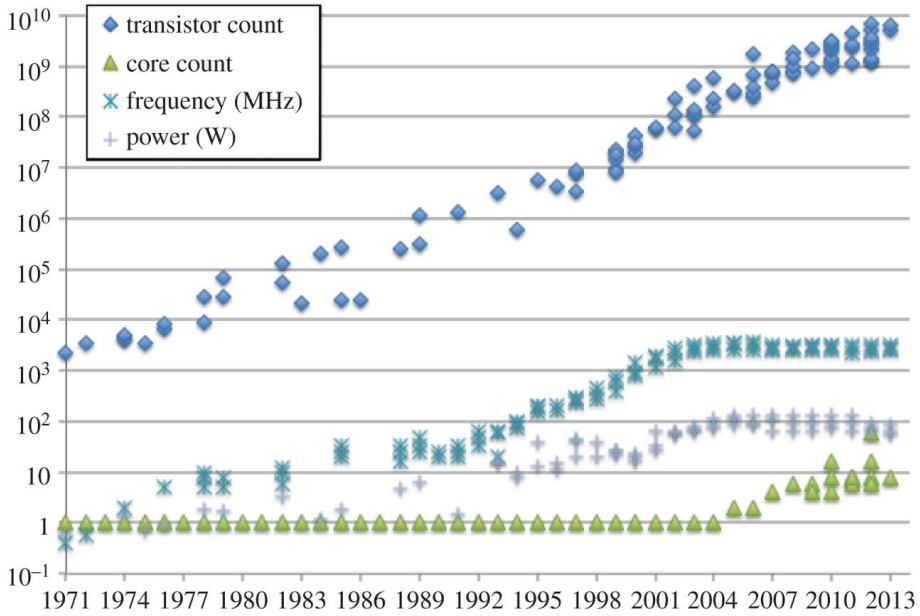


Figure 2.4: CPU Evolution. (Taken from [GR14])

To support the parallel execution of programs, a program must be mapped to a *unit of execution* (UE), such as a process or a thread. A process combines a collection of resources that enables the execution of program instructions. These resources include virtual memory, I/O descriptors, a run-time stack, signal handlers, user and group IDs, and access control tokens. As a result,

a process is a *heavy-weight* unit of execution with its own address space. In contrast, a thread is a *light-weight* UE that is associated with a process and shares the process's environment. This enables threads to perform context switches much more efficient compared to processes.

A thread or a process must be scheduled on a logical or a physical processor core. Logical cores enable a CPU to execute instructions from different threads concurrently. This technique is called *simultaneous multi-threading* (SMT) and it shares the execution resources of a CPU among different threads. Therefore, each logical core exhibits its own architectural execution state of a program including the content of its data, segment, control, and debug registers. On the other hand, all logical cores share the same execution engine and memory hierarchy. As a consequence, SMT exploits parallelism of concurrently running application threads on one CPU using out-of-order instruction scheduling to maximize the utilization of all CPU capacities.

In contrast to SMT, *multi-core CPUs* contain multiple physical cores on one socket. However, each of these cores might use SMT to provide logical cores. In terms of hardware consumption, SMT requires less physical space compared to multi-core CPUs. Especially, the complex dispatch logic, the functional units as well as the caches occupy more physical space. Finally, a computer may contain multiple CPUs on different sockets, so-called *Simultaneous Multiprocessing* (SMP).

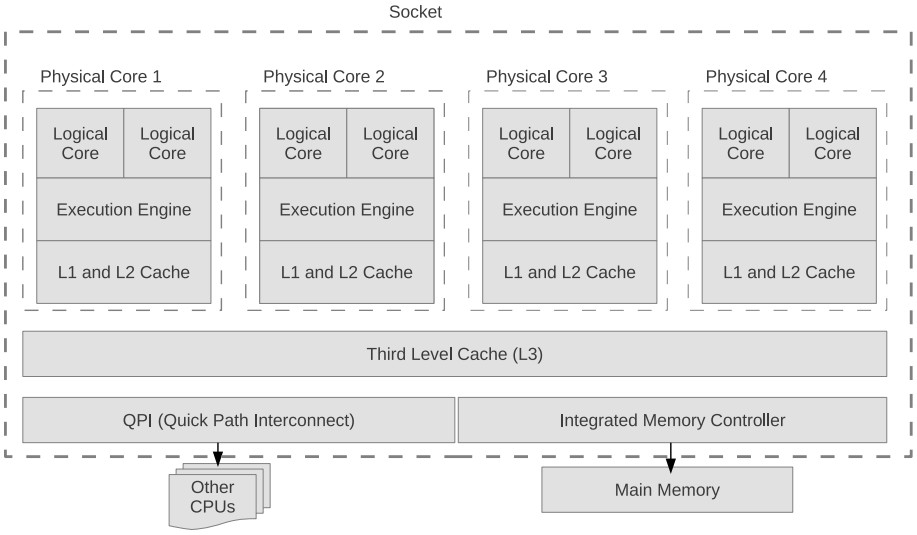


Figure 2.5: I7 Architecture. (Taken from [Int12b])

Figure 2.5 shows the architecture of an Intel I7 CPU. It contains four physical cores and eight logical cores. Each physical cores has its own L1 cache, L2 cache, and execution units. All cores share a common L3 cache, the *Quick Path Interconnect* (QPI) interface to transfer and receive data from

other CPUs, and the *Integrated Memory Controller* to transfer and receive data from main memory.

2.3 SIMD Instructions

Michael J. Flynn [Fly72] published a classification of computer architectures in 1966. This classification divide computer systems depending on their concurrent instruction streams and data streams. Table 2.3 summarizes this classification. A *SISD* computer system processes a single instruction stream on a single data stream. This system represents early computers with a sequential execution pattern without parallelism. A *SIMD* computer system uses a single instruction stream to processes multiple data streams. A *MISD* computer system executes multiple instruction streams on a single data stream. This system represents an uncommon architectures which is seldom used. Finally, a *MIMD* computer system executes multiple instruction streams on multiple data streams. This systems represent the today common multi-core superscalar processors and distributed systems.

	Single Instruction	**Multiple Instruction**
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Table 2.3: Flynn's Taxonomy [Fly72].

Following Flynn [Fly72], SIMD represents a form of parallelism, so-called *data parallelism*. Instead of applying different instructions to the same data in parallel, SIMD executes the same instruction on different data in parallel. This form of processing groups data into *vectors* and thus is called *vector processing*. Vector processing is often used by scientific, imaging, video, and multimedia applications [MSM04]. In particular, it is extensively exploited by supercomputers. As a consequence, almost all modern CPU architectures support SIMD instructions, e. g., SPARC (VIS), x86 (MMX/SSE/AVX), POWER/PowerPC (AltiVec) and ARM (NEON) [Int12b, AMD13, Pat15].

Two parameters determine the degree of parallelism for SIMD execution. First, the *SIMD bandwidth* as the size of a SIMD register determines the number of bits that can be processed in parallel by one instruction. Second, the *data type* defines the size of one data item in a SIMD register. Each data item resides in a so-called *segment* inside the register. Therefore, the data type impacts the number of parallel processed data items. For example, a 128-bit SIMD register processes sixteen 8-bit or eight 16-bit data items with one SIMD instruction.

In Figure 2.6, we show two SIMD registers of 128-bit length. The entire register is divided into four 32-bit segments. On register *R1*, we execute a

SIMD instruction which adds the integer value 2 to each segment value and writes the result into register *R2*.

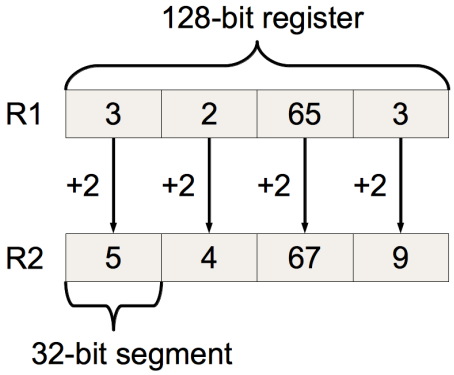


Figure 2.6: SIMD Example.

The provided SIMD instructions are chipset-dependent and differ among various processor architectures. Intel offers a wide range of arithmetical, comparison, conversion, and logical instructions [Int12b]. For example, a SIMD comparison instruction splits a SIMD register into segments of fixed size, e. g., 8, 16, 32, or 64 bits, depending on the used SIMD instruction. The comparison is performed for all segments in parallel with the corresponding segment in another SIMD register. The result of this comparison is a bitmask which is stored in a third SIMD register.

Early processors started with SIMD registers of 32 bits. With an increased number of transistors, SPARC VIS and x86 MMX doubled the bandwidth to 64 bits. Within the x86 architecture, the *Streaming SIMD Extensions* (SSE) added eight 128-bit registers. Later on, the *Advanced Vector Extensions* (AVX) widened the bandwidth to 256 bits and AVX-512 to 512 bit. Instead of widening the registers, ARM NEON processes pairs registers and treats them as one such that they may use 32 64-bit or 16 128-bit registers [Pat15]. Additionally, the segment size inside the SIMD registers changed too. Modern processors provide segment sizes of 8, 16, 32, and even 64 bits. For example, AVX supports a 4-way parallel floating-point multiply-add as a single instruction.

2.4 Cache Hierarchy

Figure 2.7 shows the memory hierarchy of modern CPUs. This hierarchy consists of multiple level of storage locations. From top to bottom, each subsequent level trades lookup speed for storage space. The fastest and smallest storage locations are registers. Registers are commonly accessed in one CPU cycle and store 64-bit of data.

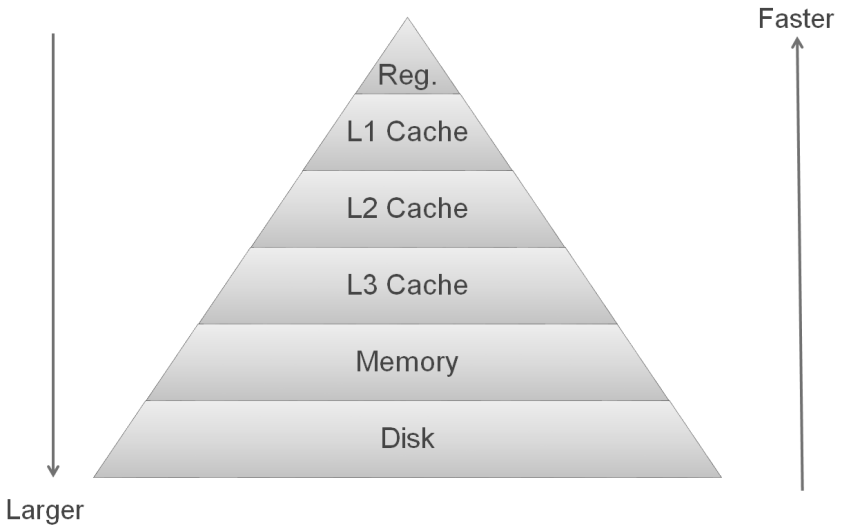


Figure 2.7: Memory Hierarchy.

The next levels form the so-called *multi-level cache hierarchy*. The *L1* cache represents the first level cache which is small, but operates near to processor speed (up to 64 KB with 4 cycles latency [Int12b]). There are commonly one dedicated L1 cache for data (L1D) and one L1 cache for instructions (L1I). The second level cache (L2) is larger, but provides slower lookup speed (up to 512 KB with 10 cycles latency [Int12b]). Commonly, there is only one L2 cache which stores data and instructions together. In a modern multi-core CPU, each core owns one L1D, one L1I, and one L2 per core as a private resource. Additionally, all cores on the same socket share a third level cache *L3*. The L3 cache is several megabytes in size with higher access latency (up to 30 MB with 40 cycles latency [Int12b]).

If more than one CPU is available in a system, i. e., a multi-socket CPU, cores from different sockets communicate via an interconnection (QPI for Intel CPUs) or via main memory [Int12b]. If the requested data item is not cached in any core on any socket, the data item must be fetched from main memory within around 100 ns. With *non-uniform memory access (NUMA)*, different cores/sockets have different access latencies to different memory locations depending on their physical distance to the responsible memory controller. Finally, if the data is not resident in main memory, it has to be fetched from disk in several milliseconds. Note that, caches work in a non-blocking manner. Thus, if a request cannot be satisfied by one cache level, it is forwarded to the next lower level. While waiting for outstanding retrievals, the cache can process other requests [Aea99].

The smallest transfer unit inside the multi-level cache hierarchy is a cache line, commonly 64 byte in size [Int12b]. When a cache line is loaded from main memory, the cache line is transferred to the CPU and additionally

placed in the cache hierarchy. It depends on the CPU policy in which level of the cache hierarchy the cache line is placed. AMD processors usually implements exclusive caches that guarantee that one cache line is placed at most in one cache, either L1 or L2 [AMD13]. Intel usually implements an inclusive policy for the L3 cache [Int12b]. Using an inclusive policy, a cache includes all cache lines from all previous cache levels in the hierarchy. Thus, the content of all caches of all cores of the same socket are guaranteed to be present in the L3 cache. This alleviates the process of detecting if another core on the same sockets holds the requested cache line in its caches. For L1 and L2 caches, Intel's XEON processor uses an intermediate approach that does not enforce inclusion, i. e., a cache line on one level is not required to be stored on another level.

Processors implement different write strategies to hold data consistent between the cache hierarchy and main memory. A *write-back* strategy updates the main memory when the cache line is replaced. The resulting inconsistent states of the same cache line in different caches of different cores and main memory is managed by a cache coherency protocol which is implemented by hardware, e. g., *MESI* [Int12b]. A *write-through* strategy immediately updates main memory when a write occurs [Smi82]. In Table 2.4, we show common sizes and latencies of caches using the example of Intel's I7 processor with an Ivy-Bridge architecture [Int12b].

Level	**Size**	**Latency**	**Physical Location**
Register	64 Bit	1 cycles	inside each core
L1 cache	32 KB	4 cycles	inside each core
L2 cache	256 KB	12 cycles	inside each core
L3 cache	6 MB	30 cycles	outside of cores
RAM	4+ GB	30 cycles + 53 ns	SDRAM DIMMs on motherboard
Disk	100+ GB	10,000+ cycles	hard disk or SSD in case

Table 2.4: I7 Memory Hierarchy.

2.4.1 Cache Architecture

A cache is characterized through its capacity, block size and associativity [HS89, SKN94]. The capacity defines the size of the cache in bytes. The block size determines how many contiguous bytes are fetched on each cache miss. This block size is also called *cache line*. On the other hand, the associativity refers to the number of unique locations in a cache at which a particular cache line may reside. In a *fully-associative* cache, the cache

line may reside at any location. In a *direct mapped* cache, the cache line resides at exactly one location. In an `N`-way cache, the cache line may reside at `N` different locations. Common caches are 8-way associative, i. e., the cache line may reside at eight different locations even if other locations are free [Int12b]. The assignment of a cache line to its position in the cache is determined by its physical address in main memory. The physical address in turn is determined by the location inside the data structure layout.

Figure 2.8 shows the access to a cache line. Logically, a cache represents a two-column table where one column represents the memory address as the key while the second column represents the cache line as the value. The *tag* part of an address is used to identify if the current entry in the cache is the required memory address. The *index* part of the address is used to determine the line inside the cache. Finally, the *offset* part is used to find the required data item inside the cache line. A cache *hit* occurs, if the tag part of the required address and the stored tag part inside the cache matches. In this case, the required data is returned as part of the corresponding cache line. Otherwise, the cache *misses* and the data request is propagated down in the memory hierarchy.

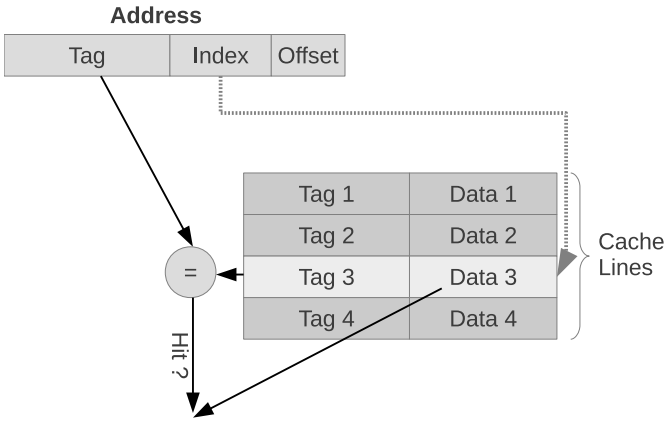


Figure 2.8: Cache Access.

Cache misses are categorized into *compulsory*, *capacity*, and *conflict misses* [HS89, SKN94]. A *compulsory miss* occurs, if an address is accessed for the first time. Random access patterns always induce several compulsory misses due to their scattered accesses. For sequential access patterns, e. g., sequential scans, the compulsory misses might be reduced due to software or hardware prefetching. A *capacity miss* occurs, if a cache failed to hold all required data at the same time. This miss occurs frequently due to a finite cache size. However, capacity misses might be reduced by increasing the temporal and spatial locality of an algorithm. On the other hand, an increased cache size will reduce capacity misses as well. A *conflict miss* occurs, if a reference hits a fully associative cache, but misses in a N-way associative

cache. In this case, the cache would be actually large enough to hold the recently referenced data, however, the associative constraints forced an eviction. Conflict misses can only be reduced by minimizing address conflicts through sophisticated mapping of data structures in main memory. By placing data at different memory locations, their assignment to different cache sets might be enforced. This technique is called *page coloring* [TDF90].

2.4.2 Caches in DBMS

Recent research has demonstrated that database workloads typically exhibit a small primary working set that resides in the cache hierarchy and a large secondary working set that resides in main memory [Jea07, Hea07c]. Several research groups investigated commercial DBMS workloads to identify the distribution between time spent on computation and time spent on waiting for data [Aea99, MDO94, AAA13, Rea98, BGB98, Kea98, Tea97, Lea98, MDO94, TS90, Eea96]. The investigated workloads can be classified as *Online Transaction Processing* (OLTP) workloads and *Online Analytical Processing* (OLAP) workloads. OLTP workloads occur in transactional databases which process a high volume of simple update/insert/delete queries [Eea96, Lea98, Kea98, TGA13]. On the other hand, OLAP workloads query large datasets using complex queries [Tea97]. In addition, some research covers both workloads [BGB98, Rea98].

Ailamaki et al. [Aea99] report that on the average, half the execution time is spent in stalls while 90% of the memory stalls are due to L2 data cache misses and L1 instruction cache misses. Other research shows similar distributions [Kea98, Rea98, RBH$^+$95]. Tözün et al. [TGA13] point out that the L1 instruction cache misses have deeper impact than data cache misses for OLTP workloads. However, most of the studies use CPUs without now commonly available L3 caches. Therefore, the L2 data cache stall time might change to L3 data cache stall time. Furthermore, Ailamaki et al. [Aea99] examined four major commercial database systems with respect to their performance on the new hardware architecture. They use *Clocks-Per Instruction (CPI)* as a metric when executing a benchmark. Even for simple database queries, the CPI values are rather high. This observation indicates, that databases are particularly ineffective in taking advantage of modern superscalar processor capabilities [Aea99, Bea05]. The major contributor for this ineffective usage are stalls.

2.4.3 Stalls in DBMS

Three different types of stalls exists in todays database systems: *data-related* stalls, *instruction-related* stalls and *resource-related* stalls. A *data-related* stall occurs, if a data item is not present on the current cache level and must be fetched from a lower cache level. To hide the resulting latency of a *data-*

related stall, the CPU either employs prefetching or tries to overlap memory accesses with other computation. Prefetching is implemented either on software or hardware level. On software level, the application may prefetch data ahead of time and may perform other useful work until the data becomes available [CGM01]. However, software prefetching requires manual computation of prefetching distances and manual insertion of explicit prefetch instruction into the code [ZCRS05, ZR04]. Additionally, a prefetch instruction is not guaranteed to be performed on commonly available processors such as Pentium 4 [Int12b]. There are restrictions that suppress a prefetch instructions, e. g., if a prefetch would incur a TLB miss [Int12b]. Furthermore, prefetching is not free of costs. Prefetching introduces overhead in terms of bus cycles, machine cycles, and resources [ZCRS05]. Worse than that, the excessive usage of prefetching may even decrease application performance due to increased resource contention [Int12b]. On hardware level, the prefetcher of modern CPUs recognizes simple access patterns, e. g., sequential scans, thus automatically prefetches data [Int12b, ZR04]. For example, a Pentium 4 processor will prefetch two cache lines for every accessed cache line that is characterized by a predictable access pattern [Hea06]. However, hardware prefetcher work inefficiently for irregular memory accesses pattern like tree traversals [Kea11]. Boncz et al. [Bea99] point out, that the CPU work per memory access tends to be small in database operations. Thus, there is a huge difference in the number of cycles needed to apply a simple selection predicate on a tuple compared to the number of cycles waiting for a tuple to be transfered from main memory. Prefetching may effectively reduce the necessary waiting time.

The second technique to hide access latency is to overlap memory accesses with other useful computations. Modern CPUs exploit this technique by introducing *out-of-order* execution. With *out-of-order* execution, the CPU may execute subsequent instructions while waiting on memory references [Rea98]. However, out-of-order execution requires enough in-progress instructions that are independent and do not incur resource-related stalls [Aea99]. Resource-related stalls occur due to unavailable execution resources inside the CPU, e. g., functional unit or register. The more data-related cache misses occur, the more instructions are required to hide the stalls. Other techniques like larger caches or improved data placement might further reduce the number of data-related cache misses [HA04]. However, techniques to reduce *data-related* cache misses do not effectively addresses *instruction-related* stalls [HA04].

In contrast to *data-related* stalls, *instruction-related* stalls cannot be overlapped and cause a serial bottleneck in the processor pipeline. If there are no other instructions available, the processor stalls and must wait until instructions are fetched from lower cache levels or main memory. Therefore, an instruction cache miss prevents the flow of instructions through the CPU and directly affects performance [HA04]. The size of the instruction cache

is subjected to the trade-off between size and latency [Hea07c]. In order to supply the CPU with instructions fast enough, the size of the instruction cache cannot be large. The main reason for small instruction caches is that a larger instruction cache will exhibit slower accesses times, which would in turn directly affect the maximum possible processor speed [HA04, ZR04]. Therefore, a relatively small instruction cache must provide the tremendous demand of independent instructions of modern CPUs to fully utilize its resources. The instruction cache performance is determined by the size of the instruction working set and the branch misprediction. To exploit the small instruction cache efficiently, database systems have to take the locality of references into account to maximize the utilization of instructions. As research by Hardavellas et al. [HA03, Hea07c] show, databases are affected by this trend in particular because they exhibit large instruction footprints and tight data dependencies. Harizopoulos et al. [HA04] show, that even the code working set of transactional operations typically overwhelms the first-level instruction cache. If the instruction cache cannot hold the entire instruction working set, a mutual eviction/load of instructions would cause *cache thrashing*.

Branch mispredictions are the second main contributor to instruction cache performance. A conditional branch instruction can lead the instruction stream to two different targets. The decision, which instruction stream will be taken, depends on the evaluation of the conditional predicate. A processor with no branch predictor would load the new instruction stream just after evaluating the branch predicate. However, such a processor would stall until the subsequent instruction stream is loaded. To overcome this stall time, modern CPUs utilize *speculative execution*.

With speculative execution, the processor guesses the outcome of a branch instruction and prefetches the predicted instruction stream. If the prediction was correct, the instruction stream is available when taking the branch and no stalls occur. However, a wrong prediction has serious performance implications. At first, a serial bottleneck occurs in the CPU pipeline and the pipeline has to be flushed. Additionally, instruction cache misses occur because the wrong instructions are prefetched, which further stalls the subsequent instruction processing. Finally, a branch misprediction induces computational overhead for computing unnecessary instructions [Int12b]. Following Ailamaki et al. [Aea99], branch mispredictions account for 20% of the total instructions retired in all their experiments.

Resource related stall time occurs, if the processor must wait for a resource to become available. Modern super-scalar processors maintain a set of different functional units and registers. By exploiting *Instruction Level Parallelism (ILP)*, modern CPUs might execute multiple instructions simultaneously on different functional units. Furthermore, considering the pipeline execution model of modern CPUs, a processor might issue a new instruction to the same functional unit each cycle [Kea11]. Although out-of-order pro-

cessing introduces some degree of freedom for dispatching instructions, the instruction pool has to contain enough different and independent instructions to fully utilize all functional units. Ailamaki et al. [Aea99] point out that CPU resource related stalls are dominated by dependency and functional unit stalls. A *dependency stall* occurs, if an instruction depends on the result of multiple other instructions that have not yet been completed; thus, serializing the instruction stream. This results in dependency stalls due to a decreased opportunity in instruction-level parallelism in the instruction pool. A *functional unit stall* occurs, if a burst of instructions tries to use more functional units than available and therefore creates contention for execution units. Following Ailamaki et al. [Aea99], functional units stall up to 5% depending on the workload. Overall, dependency stalls contribute up to 20% to the overall execution time. On the one hand, too many instructions lead to contention for the functional units. On the other hand, an insufficient number of instructions leads to under-utilization of the resources due to dependency stalls.

In summary, caches largely enhance *data-related* performance of DBMS if the primary working set of the workload fits into the cache hierarchy. If not, the performance improvements are only marginal and decrease with increasing size of the working set [Hea07c]. As long as the instruction working set of a DBMS fits into the instruction cache, the DBMS will supply the CPU with enough instructions to fully utilize its capabilities. Otherwise, instruction cache thrashing will occur, which will reduce overall DBMS performance. Finally, a DBMS must provide a high number of different and independent instructions to address resource-related stalls.

2.4.4 CPU Buffer

Besides data and instruction caches, the *Translation Lookaside Buffer* (TLB) and the *Branch Target Buffer* (BTB) impact the performance of database system significantly. Research has shown [Aea99, Bea99, ZR03], that taking these buffers into account leads to an increased application performance.

There are two types of memory addresses in modern computers. Applications refer to *virtual* memory locations. Therefore, processes and threads see the memory of a computer as a contiguous address space. To enable this view, the operating system manages the assignment of virtual addresses to *physical* memory locations. The *memory management unit* (MMU) in modern CPUs implement this assignment by translating virtual to physical addresses. This design enables a computer to provide a virtual address space to programs that is larger than the real capacity of main memory. Additionally, applications do not have to load data explicitly into memory and the memory can be better isolated between different processes. However, if the CPU tries to access a virtual address, it has to be translated into a physical page address. This translation takes time and must be performed for each

reference. To reduce the number of required translations, modern CPUs introduce a TLB cache that stores translations of the most recently accessed addresses (typically 64). Commonly, processors contain one dedicated TLB cache for translating references to data pages and one for translating references to instruction pages. If a translation is already cached in the TLB, a *TLB hit* occurs and no additional translation is necessary. However, if the translation is not cached, a *TLB miss* occurs and the translation has to be computed. The computed translation is then stored in the TLB and evicts another translation if the TLB cache is full. The translation includes main memory accesses to the operating system page directory and tables and some computation [ZR04]. The more pages an application accesses, the higher is the probability of a TLB miss. Cieslewicz and Ross [CR08] show the impact of TLB misses for partitioning data into groups. In general, partitioning requires write access to many different memory locations. If an application uses more than 64 pages, e. g., performing a random access pattern, misses occur and the behavior is analogous to the behavior of caches. Thus, random access to different pages in main memory exhibits worst TLB performance and reduces DBMS performance. For this reason, Boncz et al. [Bea99] propose a radix join which exploits partitioning and takes the TLB parameter into account to improve join performance. Another approach to amortize TLB misses and cache misses overhead over time is by processing data in batches [ZR03].

The *Branch Target Buffer* (BTB) stores the target of recently executed branches. If a branch address is already in the BTB, the buffer activates a branch prediction algorithm that predicts the branch target based on previous branching history. On the other hand, if the branch was never executed before, a static prediction is applied. The static rule predict that backward branches, e. g., at the end of a loop, are taken and forward branches, e. g., a if-then statement, are not taken [Aea99]. Ailamaki et al. [Aea99] report a 50% BTB miss rate on average for database algorithms. Therefore, the branch prediction unit is only used half of the time. The remaining predictions use the simple static rule. Additionally, an increasing BTB miss rate leads to an increased branch misprediction rate. If a branch prediction was wrong, pipeline stalls occur and the DBMS performance decreases because wrong instructions were prefetched and executed. Therefore, branch misprediction stalls are tightly connected to instruction stalls. For example, a Pentium 4 with a 20 stages deep pipeline suffers a minimal branch misprediction penalty of at least 20 cycles [ZR04]. Ailamaki et al. [Aea99] show that first-level instruction cache performance follows the behavior of the BTB cache.

Overall, the BTB is exploited efficiently if an algorithm induces as few branches as possible and the induced branches exhibit a repetitive access pattern. In addition, a small number of branches improve the spatial locality of applications by maximizing the utilization of already loaded instructions

[HA04]. Finally, a large number of branches requires a large branch history. Common BTBs store branch histories between 512 and 4000 branch targets [ZR04, Int12b]. When this capacity is exceeded, cache thrashing occurs. Ross [Ros02] points out that data-depended relational operations, e. g., a selection operator with a medium selectivity, are hard to predict and thus significantly reduce DBMS performance. On the other hand, selection operators with a low or high selectivity are simple to detect and exhibit a low misprediction rate.

2.4.5 Locality

In the previous section, we covered multi-level cache hierarchies of modern CPUs and how they impact database performance. Research in this area has proposed cache-conscious algorithms and data placement techniques to reduce cache-related stalls in database systems. To exploit the capabilities of multi-level cache hierarchies efficiently, algorithms have to take their specific caching behavior into account. There are two major aspects that determine the performance of an algorithm on a multi-level cache hierarchy. In Section 2.4.5.1, we cover *temporal locality*, i. e., how an algorithm accesses data. In Section 2.4.5.2, we cover *spatial locality*, i. e., how data is placed inside the memory hierarchy. Shatdal et al. [SKN94] introduce five general techniques to improve the cache performance of an algorithm. Three techniques try to adjust the working set size of an algorithm in relationship to the cache size. With *blocking*, the algorithm is modified to reuse chunks of data that fit into the cache [Pea01]. With *partitioning*, the entire data set is split into portions that fit into the cache. The *extraction of relevant data* reduces the cache space occupation of each tuple, thus only loading required attributes. The two remaining techniques are *loop fusions* and *data clustering*. Loop fusion merges loops which access the same data structure. In contrast, data clustering stores concurrently accessed data together.

2.4.5.1 Temporal Locality

Temporal locality, also called locality by time, refers to the observation that currently processed data are very likely to be used again in the near future [Smi82]. By storing these data items in the cache between two subsequent accesses, the reusability is increased. Ideally, a data item is loaded once, processed many times and can be evicted afterwards without any further reuse. However, due to finite cache capacity, there is a high possibility that a data item is evicted between two subsequent accesses. In general, the content of a cache depends on the access pattern, i. e., the order of data accesses over time. The longer the time span between subsequent accesses to the same data item, the higher the probability that the data item is replaced by another one. In the worst case, each subsequent data access requires to reload a data item

from a lower cache level or even from main memory. As a result, database systems with a low processing time per data item become quickly memory bound [Bea99]. In this case, the database system spends most of its time waiting for data instead of performing computation. In order to alleviate this bottleneck, a database system has to save memory bandwidth by exploiting data access locality in the multi-level cache hierarchy. Therefore, a database system has to perform as much processing as possible, preferably the entire processing, before replacing a data item. To summarize, the reuse of data in caches is mainly determined by the access pattern that in turn is defined by database algorithm.

Database algorithms define the computation performed on the data. They are implemented as operators in a query plan. The cost of an operator depends on the amount of data that has to be processed and its processing complexity. Database systems use cost models to estimate these costs. In general, a query plan that has to process less data will consume less resources and take less time to be evaluated [Man02]. Manegold et al. [Man02] propose such a cost model that takes the multi-level cache hierarchy of modern CPUs into account. The memory access costs are estimated by the number of cache misses at each individual cache level multiplied with the individual cache latency. The total cost over all individual levels represents the total cost of an operator on a given multi-level cache hierarchy. To determine the number of cache misses at each cache level for a given operator, Manegold et al. [Man02] define six basic access patterns. These basic patterns distinguish between sequential and random access to data items. In this model, a data traversal is modeled by one or two input cursor and one output cursor. The movement of the cursor represents the access pattern and thus the number of accesses to each data item. The input cursor either accesses each data item only once (single), predictable and multiple times (repetitive), or unpredictable times (random). By combining the basic patterns, Manegold et al. [Man02] describe the memory access patterns of database operations. To take hardware characteristics into account, they perform measurements using a special tool. The output of this measurements are used to parameterize the access patterns.

2.4.5.2 Spatial Locality

Spatial locality, also called locality by space, refers to the observation that data items adjacent to the currently accessed data item are likely to be accessed in the near future [Smi82]. To exploit such behavior, the adjacent data items must be loaded before they are accessed. This requires prefetching of data besides the current data item. Prefetching on hardware level is implemented at different granularities. A cache line is the smallest transfer unit inside the multi-level cache hierarchy. Common cache line sizes range from 16 bytes to 64 bytes [Man02]. Considering typical data types of sev-

eral bytes, there is a high probability that a cache line contains more than one data item. In such a case, the first reference to a data item suffers a cache miss penalty and triggers a cache line load. However, following references to other data items in the same cache line will introduce no additional cache misses if they are accessed before eviction. The next larger unit of prefetching are cache lines. Modern CPUs prefetch cache lines if they detect a predictable access patterns. The largest unit of prefetching occurs in modern DRAM chips by using *Extended Data Output (EDO)* [Man02]. EDO transfers the requested data as well as data at subsequent addresses. Therefore, the memory access to subsequent addresses introduces no additional memory transfer delays. However, only a sequential access pattern benefits from prefetching efficiently. By accessing data items sequentially, each adjacent data item is already loaded without any transfer delays. In contrast, a random memory access pattern will probably exceed the prefetch distance. In general, a sequential access pattern will be faster than a random access pattern due to better cache line utilization and exploitation of prefetching capabilities. However, Boncz et al. [Bea99] point out that the performance of an algorithm using a random memory pattern can be improved if its accessed subset fits into the cache [Bea05, SKN94, Bea99]. To summarize, spatial locality increases the chance of a cache hit for future references that are close to the recently accessed data item. Additionally, prefetching strongly impacts the effectiveness of spatial locality.

In databases systems, some operators such as a relational selection exhibit a sequential access pattern. A selection starts with the first tuple of a relation and proceeds by processing each adjacent tuple successively. Thus, the first access suffers a cache miss penalty and triggers a cache line load. The number of additional tuples per cache line is determined by the tuple size and data placement of the accessed data structure. A tuple size smaller than a cache line size leads to a high cache line utilization if each tuple in the cache line is accessed before the cache line is evicted. However, a tuple size equal or larger than cache line size prevents spatial locality inside the cache line. Research by Boncz et al. [Bea99] shows, that database performance decrease if a tuple spans multiple cache lines because each access results in at least one cache miss. Therefore, prefetching will increase spatial locality as long as the tuple size does not span across its prefetch distance.

Database systems are able to exploit spatial locality by placing subsequently used data next to each other in memory. The *best* data placement strategy has been discussed extensively over the last decade [Zea08, Bea99, AMH08, Hea06, Sea05, Hea06]. In summary, there is no strategy that provides optimal performance over all possible workloads. However, two different storage models are commonly used in commercial database systems and research prototypes. The *N-ary storage model* (NSM) stores tuples of a relation consecutive in main memory next to each other [Zea08]. Thus, each tuple stores all its attribute values in consecutive in memory locations

following the table schema. In contrast, the *decomposition storage model* (DSM) stores the columns of a relation consecutive in main memory [CK85]. Therefore, DSM distributes the attribute values of a tuple over main memory as opposed to the NSM model where the attribute values are stored consecutive in main memory. The superior of NSM or DSM strongly depends on the database workload. For example, the DSM model is advantageous for scanning few columns entirely. In this case, only the required columns are scanned and spatial locality can be exploited. Furthermore, the bandwidth demands are reduced for queries that access many tuples but not all columns. In contrast, a full column scan in an NSM model will result in poor cache line utilization because the entire tuple is loaded but only few attributes are used. All other attributes are loaded without any usage. Another example is a workload that entirely accesses a single row. In this workload, the NSM model is advantageous because it loads and accesses the entire tuple. In contrast, DSM model has to reconstruct the tuple by using several random memory accesses. Additionally, each loaded cache line contains only one required attribute value. All other attribute values of other tuples remain unused. Zukowski et al. [Zea08] point out, that the data set size in relationship to the cache sizes and the usage of SIMD also impact the performance of different placement strategies. Furthermore, they propose an on-the-fly transformation that switches the data layout before and after performing operations to the best layout. Harizopoulos et al. [Hea06] show, that the tuple size and the number of accessed columns are mainly influence the performance of NSM and DSM. NSM performs better for lean relations with small tuple sizes and for CPU-constrained environments. With increasing tuple sizes, the DSM model exhibit superior performance. Additionally, there is a crossover point when more than 85% of the tuple size is accessed. In this case the NSM outperforms the DSM.

In this chapter, we introduced the hardware characteristics of modern CPUs. We showed how algorithms in general and database system in particular might exploit the huge capabilities of modern CPUs. However, we also highlighted the difficulties and trade-offs between different CPU characteristics. In the next chapters, we provide new approaches for database systems to exploit modern CPUs efficiently.

Chapter 3

Exploiting SIMD for Query Execution

In this chapter, we present our approach to accelerate the processing of tree-based index structures by using Single-Instruction-Multiple-Data (SIMD) instructions. We adapt the B^+-Tree and prefix B-Tree (trie) by changing the search algorithm on inner nodes from binary search to k-ary search. The k-ary search enables the exploitation of SIMD instructions, which are commonly available on most modern processors today. The main challenge for using SIMD instructions on CPUs is their inherent requirement for consecutive memory loads. Therefore, data for one SIMD load instruction must be located in consecutive memory locations and cannot be scattered over the entire memory. The original layout of tree-based index structures does not satisfy this constraint and must be adapted to enable SIMD usage. Thus, we introduce two tree adaptations that satisfy the specific constraints of SIMD instructions. We present two different algorithms for transforming the original tree layout into a SIMD-friendly layout. Additionally, we introduce two SIMD-friendly search algorithms designed for the new layout.

Our adapted B^+-Tree speeds up search processes by a factor of up to eight for small data types compared to the original B^+-Tree using binary search. Furthermore, our adapted prefix B-Tree enables a high search performance even for larger data types. We report a constant 14 fold speedup and an 8 fold reduction in memory consumption compared to the original B^+-Tree. This work was published in [ZFH14].

The rest of this chapter is organized as follows. In Section 3.1, we present previous work in research field of SIMD exploitation for index structures. After that, we present our contribution in this research field. First, we introduce our ideas in Section 3.2. Second, Section 3.3 covers the exploitation of SIMD for comparing two elements. In particular, we will discuss the SIMD chipset extension of modern processors and their opportunities. Then, we outline the *k-ary search* in Section 3.4 as the foundation for our work.

Sections 3.5 and 3.6 cover our adaption of a B^+-Tree (called *Segment-Tree*) and prefix B-Tree (called *Segment-Trie*) using k-ary search. The evaluation of our tree adaptions is presented in Section 3.7. Finally, we summarize and mention possible future work in Section 3.8.

3.1 SIMD for Databases

In this section, we present previous work on SIMD exploitation in databases. In Section 3.1.1, we show how SIMD was exploited in different database components to speedup processing. After that, we present previous work on SIMD for speeding up index traversal in Section 3.1.2. This thesis contributes a new technique to exploit SIMD on tree-based index structures.

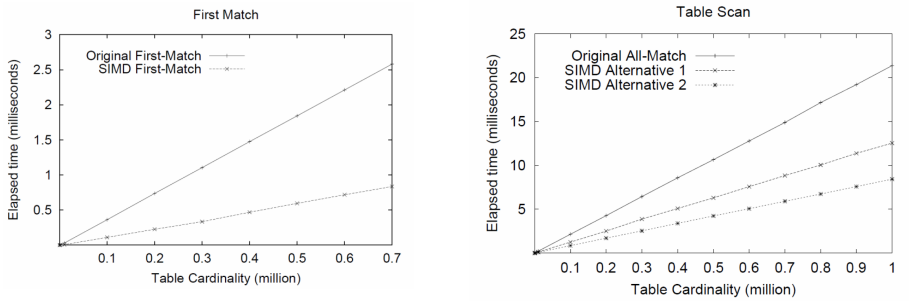
3.1.1 SIMD Usage in Databases

Databases exploit SIMD instructions to enable intra-instruction parallelism (also called *vectorization*). This form of parallelism reduces the number of executed instructions by allowing each instruction to consume more data. The SIMD register size increases over the latest micro-architectures form 128-bit to currently 512-bit [AVX08]. The possible speedup of SIMD instructions depend on the data-type that is processed. For example, processing 32-bit integers in a 256-bit SIMD register enables us to process eight values in parallel. The possible operations include various comparison, arithmetic, shuffle, conversion and logical operations [ZR02]. Slingerland et al. [SS00] conduct a detailed comparison of SIMD technology on different architectures. Zhou et al. [ZR02] point out, that database operations implemented with SIMD instructions can mostly omit conditional branch instructions. As a consequence, the significant performance penalty of branch mispredictions can be reduced.

One way to introduce vectorization in databases would be to use vectorizing compiler. These compiler detect opportunities for vectorization automatically and transform scalar source code into vectorized equivalences. For example, Intels *icc* compiler provide sophisticated automatic-vectorization capabilities. Using *Guided Auto Parallelization (GAP)*, Intels compiler can help analyze source code and generate advice on how to incorporate vectorization. Furthermore, Intel provides several advisory programs to optimize source code by using threads or vectorization [Int17b]. However, Zhou et al. [ZR02] showed that database operations cannot be automatically vectorized. In the same way, Polychroniou et al. [PRR15] point out, that none of their implemented database operations could exploit auto-vectorization. There are three main reasons for this behavior. First, *stylistic issues* could prevent auto-vectorization by using global structures or moderately complex expressions. Second, *hardware issues* such as data alignment inhibit vectorization that has to load data from aligned boundaries. Third, *complexity*

issues, such as the use of function calls or non-assignment statements in a loop are difficult to optimize. In sum, non of the basic database operations implemented by Zhou et al. [ZR02] and Polychroniou et al. [PRR15] could be automatically vectorized by Intels icc compiler [Int17b].

In the context of databases, Zhou and Ross summarize possible applications of SIMD instruction to implement database operations [ZR02]. They address sequential scans, aggregations, index operations, and joins. In the following, we outline their main insights. Zhou et al. [ZR02] implement three different scan methods commonly used in databases. First, a *first-match* scan returns the first element that satisfies a condition. Second, a *unique-match* scan returns the same result but make use of the fact that there could be only one match in the data set. Third, a *all-matches* scan returns a list of qualifying tuples. Figure 3.1 shows the improvements of a selection using SIMD compared to a selection using only scalar instructions. As shown in Figure 3.1a, the first-match scan improves the performance up to a factor of three and the performance improvements increase with larger data sets. In Figure 3.1b, the all matches scan was implemented as a branching variant (SIMD Alternative 1) or a branching-free variant (SIMD Alternative 2). For an all-matches table scan, the performance improvement originates due to branch misprediction improvements and partially due to increased parallelism. Compared to the first-match algorithm, the relative performance improvements are smaller because storing the result in an additional list requires additional memory accesses and cannot be done in parallel.



(a) Unique Match.

(b) Table Scan.

Figure 3.1: Selection Methods. (Taken from Zhou et al. [ZR02])

Based different scan variants, Zhou et al. [ZR02] implement four aggregation functions: *Sum, Count, Min,* and *Max*. They show, that the elimination of conditional branches speeds up the processing of aggregations significantly.

As another field of application, Zhou et al. [ZR02] describe how SIMD instruction can be employed to efficiently search internal nodes and leaf nodes in a tree structure. The first approach improves the binary search by expanding the number of elements in one iteration step. Instead of comparing

one separator with the search key, they use the entire SIMD bandwidth. As a result, they include elements that are located besides the separator. The second approach is a sequential search using the entire SIMD bandwidth. Instead of comparing one element at a time, the second approach compares as many elements as fit into one SIMD register and proceed in a step-wise manner. The third approach combines both approaches in a so-called *hybrid search*. This search groups data set into segments and apply binary search until the correct segment is located. After that, the located segment is scanned sequentially. Figure 3.2, we show the search results presented by Zhou et al. [ZR02]. As shown, the hybrid search performs best for node sizes larger than 200. In contrast, sequential scan methods slow down for larger node sizes. However, for small node sizes up to 200 keys, sequential scan methods exhibit similar performance as the hybrid search.

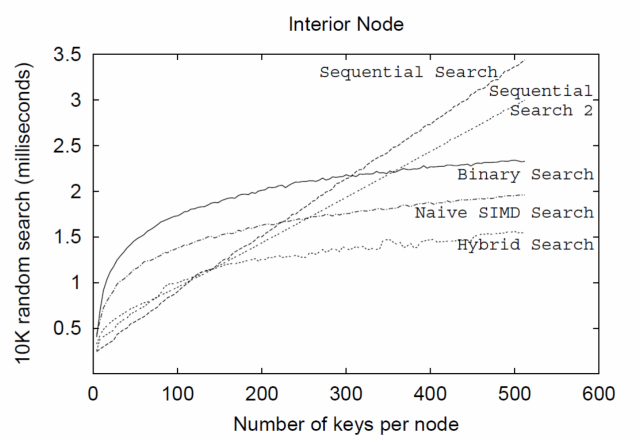


Figure 3.2: Performance of different search methods. (Taken from Zhou et al. [ZR02])

In this chapter, we will contribute to this research field by proposing a different search algorithm using SIMD instructions. In contrast to Zhou et al. [ZR02], our approach is based on k-ary search that reorders the sorted list of elements and increases the number of separators. Additionally, the k-ary search supports a distance between two separators that is wider than SIMD bandwidth.

Finally, Zhou et al. [ZR02] showed that a nested loop join could also benefit from using SIMD instructions. They provide three different implementations of a join algorithm using SIMD. First, the *duplicate-outer* algorithm fetches one join key from the outer relation and duplicate it into each segment of a SIMD register. The number of duplicates is defined by $\frac{SIMD-Register-Size}{Data-Type-Size}$. In a next step, the inner loop scans through all keys in the inner relation, load them into a SIMD register, and compare both registers to find a match. Second, the *duplicate-inner* algorithm reverses the order

such that the outer relation is scanned for each inner relation key. Third, the *rotate-inner* algorithm loads one SIMD register with inner and one with outer keys. Then, it compares these two registers n times. Between these comparison operations, the inner register is rotated by one word. Zhou et al. [ZR02] showed in their evaluation, that duplicate-inner is the slowest algorithm because it produces the most key duplicates. Overall, the rotate-inner algorithm is the fastest. Furthermore, no SIMD algorithm show a significant number of branch mispredictions.

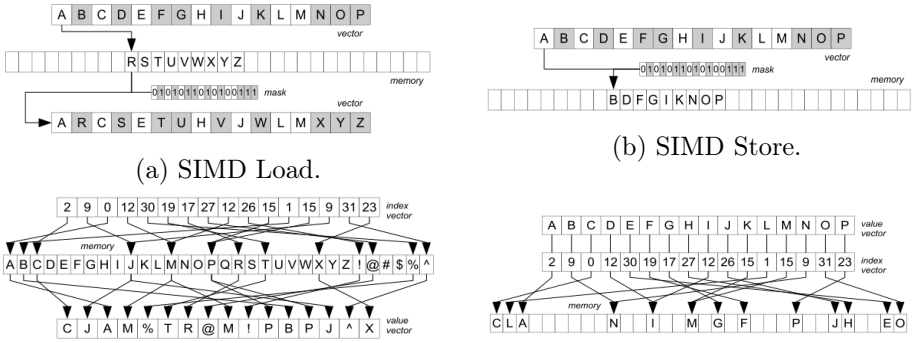
In the context of databases, SIMD was examined for sorting. Chhugani et al. [Cea08] present an implementation of *MergeSort* using SIMD instructions. Furthermore, they examine parallel SIMD usage in modern chip multiprocessor (CMP) architectures. Inoue et al. [Iea07] introduce *AA-Sort*, a parallel sorting algorithm for multi-core CPUs using SIMD instructions. Both approaches focus on sorting a list of elements. Additionally, Schlegel et al. [SWL11] explore the SIMD usage for sorted-set intersection.

Landra et al. [Lea12] show, that algorithms for text/string processing can benefit from SIMD instructions as well. They remark, that an improved string algorithms are able to accelerate applications that extensively use indexing or searching. Additionally, Schlegel et al. [SGL10] exploit SIMD instructions for fast integer compression. Furthermore, Ross [Ros07] proposed SIMD instructions for hash probing.

Polychroniou et al. [PRR15] takes the exploitation of SIMD instructions for database operations one step further by defining four basic SIMD patterns. Based on these patterns, they construct complex operators such as hash tables and partitioning algorithms. Furthermore, they combine these pattern to advanced relational operators such as sorting or joins. Figure 3.3 presents these SIMD pattern. The input values are stored in arrays and the element selection is based on a bit mask. In Figure 3.3a and Figure 3.3b, the SIMD load and store operations are shown. Based on a bitmask, an element is either stored or loaded from position i if the mask value at position i is one. Otherwise, this element is omitted.

Whereas SIMD load and store instructions operate on continuous memory locations, the gather and scatter operations shown in Figure 3.3c and Figure 3.3d load or store elements from random memory locations. For a gather instruction, an input array specifies the array elements that are loaded. For a scatter operation, the input vector specifies at which position in the array the values are stored. Note that, gather operations on CPUs are only available on the latest micro-architectures [AVX08]. In contrast, scatter instructions are not supported on CPUs and thus can be used only on GPUs or on MIC-architectures like the Xeon-Phi [Tea16]. Alternatively, Polychroniou et al. [PRR15] suggest to replace missing instructions by slower available instructions.

Based on these SIMD patterns, Polychroniou et al. [PRR15] implement hash build and probing using multiple hashing schemes. For partitioning,



(a) SIMD Load.

(b) SIMD Store.

(c) Gather.

(d) Scatter.

Figure 3.3: Basic SIMD Pattern. (Taken from Polychroniou et al. [PRR15])

they provide implementations for histogram generation as well as three partitioning functions: radix, hash, and range. Finally, they combine these patterns to implement radix sort and multiple hash join variants. Their evaluation showed, that SIMD improves the throughput of these operators significantly.

Finally, GPUs provide an extended SIMD functionality. To exploit their enormous computing power for DBMS, GPUs represent a strong research field in the database community [Hea, Gea04, Bea09, Gea06, Aea09]. GPUs able to massively parallelize SIMD computations on independent data items. The challenge for GPU computing is to supply the GPU with the required data. In case of a computation-bound application, this might succeed. However, data must be shipped to the GPU and the result must be shipped back to the CPU for evaluation. In data-intensive applications like databases, this transport will probably become a bottleneck which prevent an efficient GPU utilization [Kea12]. As an alternative, CPU vendors integrate SIMD register and instructions sets in modern CPUs. These instructions enable SIMD processing inside the CPU but with a much smaller level of parallelism.

3.1.2 SIMD Usage for Indices

In this section, we present previous work about SIMD exploitation for indices. Kim et al. [Kea10] introduce *FAST*, a binary tree that is optimized for architectural features like page size, cache line size, and SIMD bandwidth of the underlying hardware. They examined the impact of translation lookaside buffer (TLB) misses, last level cache (LLC) misses, and memory latency on CPU and GPU. Furthermore, Kim et al. [Kea10] exploit thread-level and data-level parallelism on both CPUs and GPUs. They point out, that tree size and the size of the LLC impacts the usability of CPU or GPU for index processing. In sum, they conclude that tree processing is computation bound on small trees which fit into LLC and bandwidth bound on trees larger than

LLC size. Compared to our approach using k-ary search for tree traversal, they use an additional *lookup table* to evaluate the bitmask and navigate to the next child node. The data layout also differs between our tree variants using k-ary search and the adapted binary search by Kim et al. [Kea10]. They divide the tree in sub-trees to create a layout optimized for specific architectural features. In contrast, our approaches use k-ary search and our data layout is determined by breadth-first or depth-first search. They report a 5X (CPU) and 1.7X (GPU) performance improvement compared to the best previously reported algorithms on the same architectures.

Based on *FAST*, Yamamuro et al. [Yea12] introduce the *VAST-Tree*, a vector-advanced compressed structure for massive data tree traversal. By applying different compression techniques to different node levels, they achieve a more compact tree with higher traversal efficiency.

Leis et al. [LKN13] introduce the *Adaptive Radix Tree* as an *ARTful* index for main memory databases. The ART tree uses four node types with different capacities depending on the number of keys. However, this approach uses SIMD instructions only for the search in one node type and for at most 16 keys. In comparison, our approach uses SIMD instructions independent of the number of keys and for all node sizes.

Graefe and Larson summarized several techniques for improving cache performance for B-Trees [GL01]. Furthermore, Bender et al. [BDFC00] introduce a cache oblivious B-Tree and a cache oblivious string B-Tree [BFCK06]. Rao and Ross introduce two cache conscious tree structure, the *Cache-Sensitive Search Trees (CSS-Tree)* [RR99] and the *Cache Sensitive B^+-Tree (CSB+-Trees)* [RR00]. These tree variants construct a tree such that keys are placed as cache-optimized as possible in terms of spatial or temporal locality. They differ in terms of knowing the main parameters of the memory hierarchy, i. e., they are cache *conscious*, or running best on an arbitrary memory hierarchy, i. e., they are cache *oblivious*. Besides these differences, all tree variants increase cache line utilization by changing the tree layout. In a similar way, our approach changes the tree layout to enable SIMD usage for tree traversal. As a result, our layout modification increases cache line utilization as well. At first, our approach maximizes cache line utilization by sorting keys such that separator keys are placed next to each other. Therefore, we compare k separators in parallel instead of two in case of the commonly used binary search. Second, our approach reduces the number of comparison operations inside the node from $log_2 n$ to $log_k n$. The decreased number of comparisons reduces the number of loaded cache lines. Furthermore, the number of accesses to different memory locations are reduced; thus, increasing spatial locality.

Following the idea of a *prefix B-trees* by Bayer et al. [BU77], many trie variations have been proposed. The *generalized trie* by Boehm et al. [Bea11b] exhibits the most similarities to our trie implementation using k-ary search. Both approaches partition a fix sized integer value and distribute it above

different trie levels. However, the inner node search differs. Inside one node, the *generalized trie* maps the partial key to a position in an array of pointers. A node contains one pointer for each possible value of the partial key domain. In contrast, our trie implementation performs a k-ary search in each node. Furthermore, our implementation will store the same pointer array and an additional array for all possible key representation. For traversal, our trie implementation performs the k-ary search in each node with two comparison operations for an 8-bit data type.

Finally, Boehm et al. [Bea11b] introduce a *Bypass Jumper Array* and the concept of *Trie Expansion* to speed up traversal. The *Bypass Jumper Array* is used to bypass existing trie nodes that contain leading zeros. The concept of *Trie Expansion* is applied in our optimized trie implementation.

3.2 SIMD supported Tree Operations

Since Bayer and McCreight introduced the B-Tree [BM70] in 1970, it has been adapted in many ways to meet the increasing demands of modern index structures to manage higher data volumes with an ever decreasing response time. The basic elements of every index structure are keys and their associated values. A key and its associated value form one data item. The key represents the searchable value inside the index structure and, if it exists, the associated value is returned. The value is either an alphanumeric data item, a tuple identifier, or a pointer to an item in another data structure. The B-Tree combines a fixed number of data items in a node and relate nodes in a tree-like manner. Each tree has one designated node, called *root node*, as the starting point for any traversal.

In the past, many variants of the original B-Tree evolved which differ, among other aspects, in the restrictions of allowed data items per node and the kind of data each node stores. The most widely used variant of the B-Tree is the B^+-Tree. The B^+-Tree distinguishes between leaf and branching nodes. While leaf nodes store data items to form a so-called *sequence set* [Com79], branching nodes are used for pathfinding based on stored key values. Thus, each node is either used for navigation or for storing data items, but not for both purposes like in the original B-Tree. One major advantage of the B^+-Tree is its ability to speedup sequential processing by linking leaf nodes to support range queries.

As a variant of the original B-Tree, Bayer et al. introduced the prefix B-Tree (trie) [BU77], also called digital search tree. Instead of storing and comparing the entire key on each level, a trie operates on parts of the key by using their digital representation. The keys are implicitly stored in the path from the root to the leaf nodes. When inserting a new key, the key is split into fix sized parts and distributed among the different trie levels. Because the key length and the partial key length are statically defined during initialization,

the height of a trie is invariant. The fixed height distinguishes a trie from other tree structures which grow and shrink dynamically. The fixed height of a trie changes the complexity of finding a key. Whereas finding a key in a B^+-Tree is $O(\log N)$, the worst-case time complexity of a trie is $O(1)$ for all operations, independent of the number of records in the trie. Furthermore, a trie may terminate the traversal above leaf level if a partial key is not present on the current level. Additionally, splitting keys allows prefix compression at different trie levels. Thus, a trie structure with its predefined parameters results in a more static structure compared to a dynamically growing and shrinking B^+-Tree. The existing trie-based structures are mainly used for string indexing [HZW02]. However, indexing of arbitrary data types is also possible [Bea11b].

An important performance factor for all tree structures is the number of keys per node. As one node is usually mapped onto one page on secondary storage, one page is copied by one I/O operation into main memory. Thus, I/O operations are the most time-consuming steps in processing a B-Tree. Other steps, i. e., CPU intensive computing, are usually negligible in the presence of I/O operations. With a node as the transport unit within the storage hierarchy, it is important to realize that processing will be faster the more keys fit into one node.

This observation has been true for the era of disk-based databases; it also holds nowadays for main memory based databases. That is, the bottleneck between secondary storage and main memory has been moved to a bottleneck between main memory and CPU caches [Sea07]. The link between two levels of the storage hierarchy will be the bottleneck if the node size is greater than the amount of data that can be transferred in one step. Therefore, the node size in a disk-based database is determined by the I/O block size and in a main memory database by the cache line size [RR99, RR00]. In this section, we focus on main memory databases. We assume, that the complete working set fits into main memory and exclude I/O impact at all. Thus, we focus on the new bottleneck between main memory and CPU caches.

As mentioned before, a performance increase for a tree structure might result from storing more keys in one node. An increased node size is less applicable for optimization because the node size is mainly determined by the underlying hardware; thus, matches a transfer unit inside the memory hierarchy. Furthermore, the size of a pointer and the size of data types are also hardware specific. On the other hand, the number of keys and pointers within one node are adjustable. For example, the B^+-Tree moves associated values to leaf nodes. Therefore, the branching nodes are able to store more keys which accelerates the traversal speed by increasing the fanout. Another approach for storing more keys in one node is to apply prefix or suffix compression techniques on each key [BU77]. The compression of stored pointers is also possible [Wag73]. If compression is rewarding besides the additional computational effort is an ongoing research question.

Searching inside a node has been discussed extensively as well. Suggested search strategies range from sequential over binary to exploration search [Com79]. We contribute a new search strategy for inner node search based on the *k-ary search* algorithm. This algorithm uses SIMD for comparing multiple data items in parallel [SGL09]. We adapt the B^+-Tree and the prefix B-Tree structure for the k-ary search algorithm. The adapted B^+-Tree performs well on small data types, i. e., data types that use up to 16 bits for value representation. To improve k-ary search performance for larger data types, we also adapt the prefix B-Tree. Both tree adaptations make SIMD instructions applicable for tree-based index structures in modern database systems.

In the light of this discussion the contributions of this section are as follows:

1. We adapt the B^+-Tree and the prefix B-Tree for SIMD usage by incorporating k-ary search.
2. We compare both adaptations and derive their suitability for different workloads.
3. We present a transformation and a search algorithm for a breath-first and depth-first data layout.
4. We contribute three algorithms for interpreting a SIMD comparison result.

3.3 SIMD for Comparison

In this section, we present different aspects that have to be considered when using SIMD. We refer to Section 2.3 for a basic introduction to SIMD and its characteristics. First, we introduce a sequence of SIMD instructions to compare two elements in Section 3.3.1. After that, we show how a result of a SIMD comparison can be evaluated in Section 3.3.2. Then, we summarize the costs that a SIMD comparison would induce in Section 3.3.3. Finally, we present a solution for the problem that current SIMD instructions support only signed values in Section 3.3.4.

3.3.1 SIMD Comparison Sequence

In a tree structure, the comparison result inside a node is used to navigate to the next child node. This series of comparisons may terminate in a leaf node that may contain the search key v. We use SIMD comparison instructions to speedup the inner node search in a tree structure; the most time consuming operation. Therefore, we need to compare a search key v with a sorted list of keys inside a tree node. Following Schlegel et al. [SGL09], our instruction

sequence for comparing a search key with a sorted list of keys contains five steps:

1. Load keys segment-wise into register $R1$.
2. Load search key v into each segment of register $R2$.
3. Run pairwise comparison for each segment.
4. Save the result as a bitmask.
5. Evaluate the bitmask.

Unfortunately, SIMD instructions do not provide *conditional or branching statements* [Int12b]. Since all operations are performed in parallel, there is no possibility to check individual values and branch to specific code. Therefore, the result of comparing two SIMD register is a *bitmask*. The bitmask indicates the relationship between the search key v and the list of keys. For the remainder of this section, we use the greater-than relationship for comparisons. By evaluating the bitmask, we get a position in the sorted list of keys. This position indicates the first key that is greater-than the search key v. In a tree structure, this position identifies the pointer which leads to the next child node.

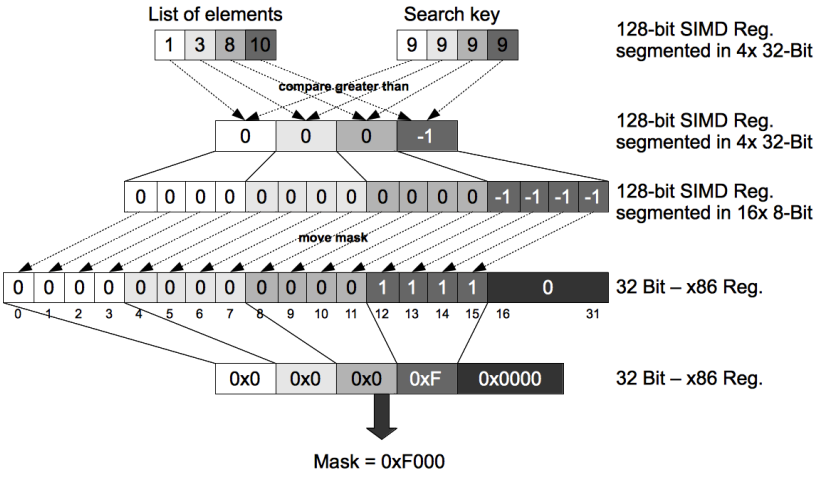


Figure 3.4: A sequence using SIMD instructions to compare a list of keys with a search key.

Our implementation of the aforementioned sequence for a 32-bit data type is illustrated in Figure 3.4. First, we load a list of keys into a 128-bit SIMD register by using the `__mm_load_si128` instruction. After that, we load the search key $v = 9$ into each 32-bit segment of a second 128-bit SIMD register with `__mm_set1_epi32`. The pairwise greater-than comparison of

SIMD instruction	Explanation
`__m128i _mm_load_si128 (__m128i *p)`	Loads a 128-bit value. Returns the value loaded into a variable representing a register.
`__m128i _mm_set1_epi32 (int i)`	Sets 4 signed 32-bit integer values to `i`.
`__m128i _mm_cmpgt_epi32 (__m128i a, __m128i b)`	Compares 4 signed 32-bit integers in `a` and 4 signed 32-bit integers in `b` for greater-than.
`__mm_movemask_epi8 (__m128i a)`	Creates a 16-bit mask from the most significant bits of the 16 signed or unsigned 8-bit integers in `a` and zero extends the upper bits.

Table 3.1: Used SIMD instructions from Streaming SIMD Extensions 2 (SSE2).

each segment is executed by `__mm_cmpgt_epi32`. This instruction compares each 32-bit segment in both input registers and outputs -1 into the corresponding segment of a third 128-bit SIMD register if the key is greater than the search key, otherwise zero. To create a bitmask as the result of the comparison, we use `__mm_movemask_epi8` to extract the most significant bit from each 8-bit segment. The sixteen extracted bits are stored in the lower 16 bits of an x86 register. Unlike a SIMD register, a x86 register provides conditional and branching statements like *if*. Table 3.1 describes the used SIMD instructions with `__m128i` as a 128-bit SIMD data type [Mic17].

3.3.2 Bitmask Evaluation

The resulting bitmask must be evaluated to determine the position of the search key within the sorted list of keys. We exploit a particular property of the *greater-than* comparison for the evaluation. When evaluating the bitmask linearly from left to right, the first key that is greater than the search key represents a *switch* point. Beyond this point, all subsequent keys are greater than the search key and thus represented with a one in the bitmask. With this property in mind, we introduce three different algorithms for bitmask evaluation. Notice, that the upper 16 bits are ignored for our evaluation. Algorithm 1 uses a loop to check if the least significant bit in each segment is set. For simplicity, we omit the case that the evaluation might terminate if we found the first greater key. In such a case, we calculate the position assuming that only greater keys will follow. c denotes the number of segments in a SIMD register that is defined by the used data type and the SIMD bandwidth. Algorithm 2 implements a switch statement for each possible bitmask of a 32-bit segment size in a 128-bit SIMD register.

Algorithm 3 uses the `popcnt` instruction to return the number of bits set in a register.

Algorithm 1 Bit Shifting

```
mask ← bitmask
c ← number of segments
position ← 0
for i = 0 → c do
    position += mask & 0x01
    mask >>= c
end for
return c − position
```

Algorithm 2 Switch Case

```
mask ← bitmask
position ← 0
switch mask do
    case 0xffff
        position ← 0
        break
    case 0xfff0
        position ← 1
        break
    case 0xff00
        position ← 2
        break
    case 0xf000
        position ← 3
        break
return position
```

Algorithm 3 Popcnt

```
mask ← bitmask
c ← number of segments
shift ← 16/c
return c − __popcnt(mask)/shift
```

By evaluating the resulting bitmask $0xF000$ in Figure 3.4 using one of the three algorithms, we get three as a result. Therefore, the first key in the sorted list of keys that is greater than the search key v is located at position three. Note, the positioning starts at zero. In a tree structure we would follow the pointer at this position.

3.3.3 SIMD Comparison Costs

The aforementioned sequence uses four different SIMD instructions. The *load* and *set* instructions load keys in SIMD register. *Set* is a composite instruc-

tion containing one load instruction for moving a value into one segment and an additional instruction for copying the value to the other segments. The *comparison* instruction compares two SIMD register and the *movemask* instruction moves the resulting bitmask into a x86 register. Modern processors of Intels *Nehalem* or *Sandy Bridge* micro-architecture are able to perform one SIMD load or comparison instruction in each CPU cycle resulting in one *cycle per instruction* (CPI) [Int12b]. However, Intel does not provide CPI information for composite instructions. In our sequence, we perform the set instruction only once to load the search key. Therefore, we exclude the set instruction from the following considerations of a simplified run-time estimation on instruction level.

We compare our SIMD sequence against the common approach using scalar instructions. First, the SIMD load and comparison instructions are as fast as similar scalar instructions operating on x86 registers. This leads to an increased *instructions per cycle* (IPC) rate because SIMD increases the number of parallel-executed instructions without introducing additional latency. However, the second step of evaluating the comparison result differs in terms of executed instructions. A sequence using scalar instructions performs conditional jumps depending on the status flags in the *EFLAGS* register. In contrast, our SIMD sequence performs one *movemask* instruction in two CPU cycles to extract a bitmask from the comparison result. After that, the bitmask is evaluated using one of the previously introduces bitmask evaluation algorithms. Section 3.7.2 will show, that despite the additional effort for bitmask evaluation, our SIMD sequence is still faster than a scalar instruction sequence.

3.3.4 SIMD on Unsigned Data Types

Current SIMD extensions of modern processors support SIMD comparison instructions only for signed data types [Int12b]. To use SIMD comparison instructions for unsigned data types, we implement a preceding subtraction by the maximum value of the signed data type. Therefore, we realign the unsigned value to a signed value. For example, the value zero of an 8-bit unsigned integer data type is realigned to -128. The value 256 is realigned to 127. With this preceding subtraction, we are able to use the signed SIMD comparison instructions for unsigned data types. As a result, the value must be realigned by insert and search operations.

3.4 K-ary Search

The *k-ary search* was introduced by Schlegel et al. [SGL09] and bases on binary search. The binary search algorithm uses the divide-and-conquer paradigm. This paradigm works iteratively over a sorted list of keys by

dividing the search space equally in each iteration. The algorithm first identifies the median key of a sorted list of keys. The median key serves as a separator that divides the search space in two equally sized sets of keys (so-called *partitions*). The left partition only contains keys that are smaller than the median key. In contrast, keys in the right partition are larger than the median key. After partitioning, the search key v is compared to the median key. The search terminates if the search key is equal to the median key. Otherwise, the binary search uses the left or right partition, depending on the greater-less relationship, as the input for the next iteration. In case of an empty partition, search key v is not in the list of elements and the search terminates. For a key count n, the complexity is logarithmic and performs $h = log_2 n$ iterations in the worst case and $h - (2^h - h - 1)/n > h - 2$ on average [SGL09]. Figure 3.5 illustrates the binary search for $v = 9$ on a sorted list of 26 keys. The boxed keys form one partition and the underlined keys show the separators.

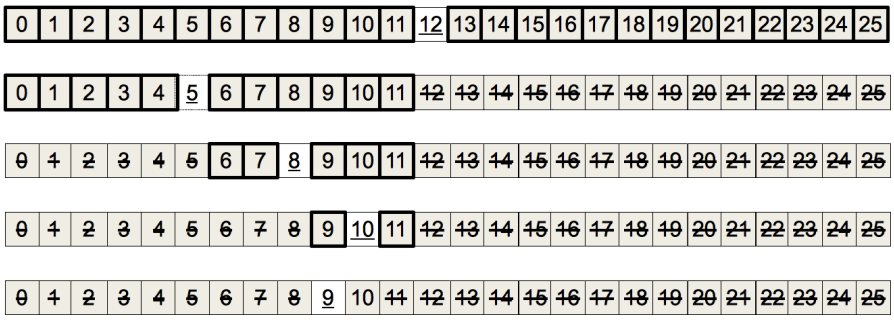


Figure 3.5: Binary search for key 9 and n = 26.

While binary search divides the search space into two partitions in each iteration, the k-ary search algorithm divides the search space into k partitions by using $k-1$ separators. We utilize our aforementioned SIMD sequence (see Section 3.3.1) to create this increased number of partitions and separators. As shown in Section 3.3.1, SIMD instructions are able to compare a list of keys with a search key in parallel. The number of parallel key comparisons depends on the data type and the available SIMD bandwidth. With parameter k, $k-1$ separator keys are compared in one iteration which increases the number of partitions to k. Figure 3.6 illustrates the same search as in Figure 3.5 now using k-ary search. The binary search compares only one key at a time with a search key; thus, producing two partitions. In contrast, the k-ary search with $k = 3$ compares two keys in parallel with a search key and divides the search space into three partitions. As a result, the k-ary search terminates after three iterations while binary search requires five iterations to find the search key. In general, k-ary search reduces the complexity to $O(log_k(n))$ compared to $O(log_2(n))$ for binary search. Assuming a commonly available SIMD bandwidth of 128-bit and a data type of 8-bit,

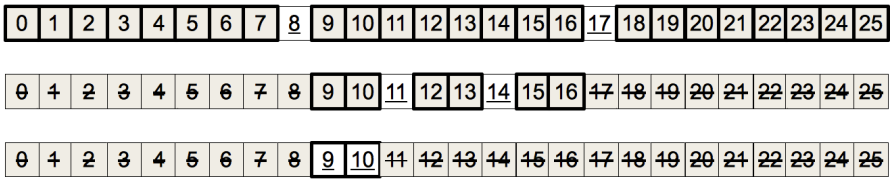


Figure 3.6: K-ary search for key 9, n = 26, and k = 3.

16 values can be compared in parallel. Using our definition of k, 16 parallel comparisons result in $k = 17$. Therefore, the number of iterations is reduced by a factor of $\frac{log_2(n)}{log_k(n)} = log_2(k) \approx 4$ for $k = 17$. The main restriction of SIMD instructions is their requirement for a sequential load of data. This requirement presupposes, that all keys that are loaded into one SIMD register with one SIMD instruction must be stored consecutively in main memory. Load or store instructions using scatter and gather operations could allow a load/store of keys from distributed memory locations [Int12b]. However, only gather instructions are supported by CPUs and only by the newest micro-architectures [PRR15].

The keys in a sorted list are placed one key next to the other in linear order as shown in Figure 3.6. Therefore, keys are placed in ascending or descending order, depending on their relationship to each other. This placement strategy is sufficient for binary search, but not amenable to k-ary search. In a linear sorted list of keys, possible separator keys are not placed in consecutive memory locations because several keys fall in between. For example, keys 8 and 17 in Figure 3.6 may be chosen as separators to partition the sorted list in three equally sized partitions. After partitioning, the separators and the search key must be compared to determine the input for the next iteration. When storing the list of keys in linear order, the separator keys are not placed next to each other in main memory and thus cannot be loaded with one SIMD instruction. To overcome this restriction, Schlegel et al. [SGL09] suggest to build a k-ary search tree from the sorted list of keys. They define a perfect k-ary search tree as: *"[. . .] every node – including the root node – has precisely $k-1$ entries, every internal node has k successors, and every leaf node has the same depth."*.

The k-ary search tree is a logical representation that must be transformed for storage in main memory or on secondary storage. For this transformation, Schlegel et al. [SGL09] propose to *linearize* the k-ary search tree. The linearization procedure transforms a sorted list of keys into a linearized k-ary search tree. Figure 3.7 summarizes the transformation process. As a result, both separator keys are placed side by side and thus can be loaded with one SIMD instruction. In Section 3.5.1, we present two algorithms that use depth-first search or breath-first search for this transformation. Figure 3.8 illustrates a k-ary search for search key $v = 9$ on a breadth-first lin-

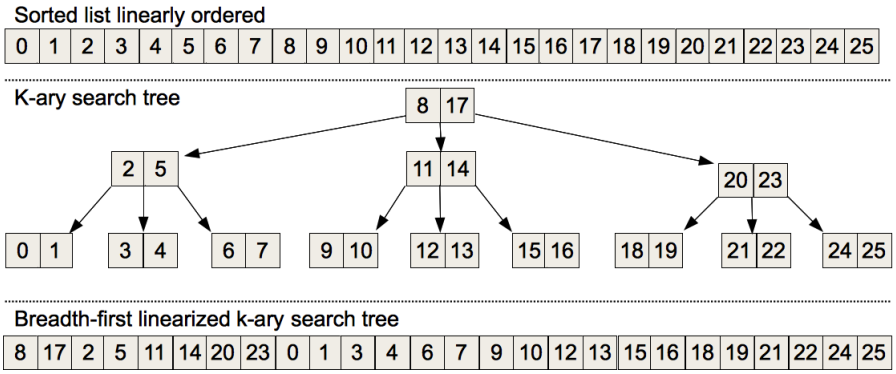


Figure 3.7: Breadth-first transformation overview.

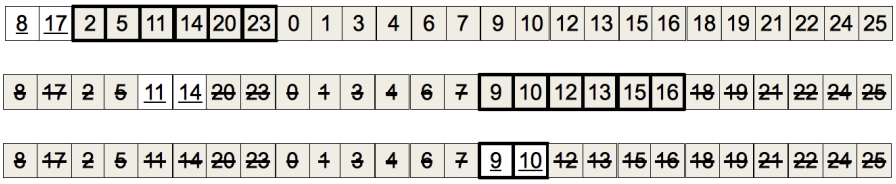


Figure 3.8: K-ary search for key 9 on a breadth-first linearized tree, n = 26 and k = 3.

earized k-ary search tree. Potentially, the introduction of gather and scatter instructions into CPUs could be exploited by k-ary search. In this case, the data set does not have to be linearized and the keys could be gathered from different array positions based on a bitmask as shown by Polychroniou et al. [PRR15]. Thus, the linear sorted order could be maintained. This change would save the costs of linearization and simplifies the insertion and deletion of keys. However, it also introduces a random memory access pattern. For large lists, k-ary search has to gather keys that are potentially located on different memory pages. As a result, cache line utilization might be reduced and sequential access to cache lines would be eliminated. Therefore, exploiting gather instructions for k-ary search would simplify tree manipulation but worsen the performance significantly.

3.5 Segmented Tree

In this section, we present our *Segment-Tree (Seg-Tree for short)* that implements the k-ary search algorithm for inner node search in a B^+-Tree. In Section 3.5.1, we adapt the basic structure and show the implications for the traversal algorithm. Section 3.5.2 presents two algorithms for linearizing a sorted list of keys. In Section 3.5.3, we address the essential ability of supporting arbitrary sized search spaces. Finally, Section 3.5.4 analyzes the performance of the Seg-Tree.

3.5.1 Using k-ary Search in B$^+$-Trees

Our Seg-Tree uses the k-ary search algorithm for inner node search in a B^+-Tree. We consider each node as a k-ary search tree. This consideration is an important aspect when updating a node. Thus, the effect of an update operation is limited to one node. This locality property eliminates the need for rebuilding the complete Seg-Tree for each update operation. The traversal across the nodes from the root to the leaves keeps unchanged compared to B^+-Trees. Furthermore, the split and merge operations in case of a node overflow or underflow are unaffected. Our approach changes the search method inside the nodes from commonly binary search to k-ary search. We store one array for n keys and one array for $n+1$ pointers inside each node.

For the rest of this chapter, let D_m define a data type with at most m bits for representing its values and $|SIMD|$ denotes the SIMD bandwidth. Furthermore, let k denote the order of the k-ary search tree with $k = \frac{|SIMD|}{m} + 1$ pointers and $k-1$ keys in each node. The number of levels in the k-ary search tree is determined by $r = \lceil \log_k n \rceil$ with n being the number of keys in the sorted list. The maximum number of keys in one node is bound by $N-1$ with $N = k^r$.

Algorithms 4 and 5 implement our sequence of SIMD instructions for comparing a search key v with a sorted list of keys (see Section 3.3.1). Based on the linearization method, either one of these two algorithms can be applied. Algorithm 5 performs a search on a breadth-first linearized list of keys in one Seg-Tree node. On each level of the k-ary search tree, $k-1$ keys are compared to a search key using SIMD instructions. The bitmask on each level is evaluated to a position using one of the bitmask evaluation Algorithms 1, 2, or 3. The resulting position will be incrementally built up during the search process and is additionally used to determine the offset for the next lower level. After the search on each level completes, a lookup into the pointer array using the returned position determines the path to the next node. Algorithm 4 is implemented in a similar way for searching on a depth-first linearized list of keys.

We refer to Figure 3.7 as a breadth-first linearized node in a Seg-Tree. The node contains $n = 26$ 64-bit keys. A SIMD bandwidth of 128-bit leads to $k = 3$. The height of the k-ary search tree is determined by $r = \lceil \log_3 26 \rceil = 3$ and the maximum number of keys $N-1$ is 26 since $N = 3^3 = 27$. Consider a search for key $v = 9$ using Algorithm 5 within this node. R denotes a SIMD register containing the search key in each segment (Line 6). C denotes a SIMD register storing $k-1$ keys. First, the algorithm determines the key pointer *keyPtr* in Line 8. Initially, *keyPtr* points to the first key in the key array (Line 3). The algorithm loads $k-1$ keys via this pointer in a SIMD register (Line 11). During the first iteration, the node (8,17) is loaded and compared to search key $v = 9$ (Line 12). The resulting bitmask is evaluated in Line 13-14. The returned position 1 is added to *pLevel* in Line 17. After

that, we determine the base pointer for the next iteration in Line 18. The base pointer refers to the left most node on the next lower level (2,5). In the second iteration, we add an offset depending on *pLevel* of the previous iteration to the base pointer in Line 8. This offsets the *keyPrt* to the desired node (11,14) and the SIMD comparison sequence in Line 10-16 returns zero. For the last iteration, we set the base pointer to the left most node on the last level (9,10). This node represents the desired node and no offset must be added. The SIMD comparison sequence in Line 10-16 returns zero. Finally, we return *pLevel* in Line 21. $pLevel = 9$ was incrementally built over all iterations and selects the first key in the Seg-Tree node that is greater than search key $v = 9$. Note, that *pLevel* is equal to the search result of a binary search on the same list of keys. Therefore, the navigation to the next Seg-Tree node is similar to the original B^+-Tree navigation. In case of a branching node, we follow the pointer at position 9 to a child node on the next level that contains values smaller or equal to search key v. For a leaf node, we would perform an additional comparison for equality to locate the associated value for search key v.

Generally, Algorithms 4 and 5 search on a linearized list of keys but returning the position as if the keys are in linear sorted order. Therefore, only the keys in the k-ary search tree must be linearized; pointers are left unchanged. Due to this important property, an update operation does not affect the pointer array. However, this property also impacts the transformation process in the next section.

Algorithm 4 Depth-First search using SIMD

```
 1: pLevel ← 0
 2: subSize ← N-1
 3: R ← set searchKey in each segment
 4: keyPtr ← pointer to first key in key array
 5: while subSize > 0 do
 6:     pLevel ← pLevel * k
 7:     subSize ← subSize − k − 1
 8:     subSize ← subSize/k
 9:     function SEARCHSIMD(keyPtr, R)
10:         C ← load k − 1 keys from keyPtr
11:         cmp ← compare C and R for greater-than
12:         bitmask ← extract bitmask from cmp
13:         position ← evaluate bitmask
14:         return position
15:     end function
16:     keyPtr ← keyPtr + k − 1
17:     keyPtr ← keyPtr + subSize * position
18:     pLevel ← pLevel + position
19: end while
20: return pLevel
```

Algorithm 5 Breadth-First search using SIMD

```
1: pLevel ← 0
2: lvlCnt ← 1
3: keyPtr ← pointer to first key in key array
4: nextBasePtr ← keyPtr
5: endPtr ← keyPtr + key count
6: R ← set searchKey in each segment
7: while nextBasePtr < endPtr do
8:     keyPtr ← nextBasePtr + pLevel ∗ (k − 1)
9:     pLevel ← pLevel ∗ k
10:    function searchSIMD(keyPtr, R)
11:        C ← load k − 1 keys from keyPtr
12:        cmp ← compare C and R for greater than
13:        bitmask ← extract bitmask from cmp
14:        position ← evaluate bitmask
15:        return position
16:    end function
17:    pLevel ← pLevel + position
18:    nextBasePtr ← nextBasePtr + lvlCnt ∗ (k − 1)
19:    lvlCnt ← lvlCnt ∗ k
20: end while
21: return pLevel
```

Search Algorithms 4 and 5 perform one comparison operation on each k-ary search tree level. In contrast, binary search has the possibility to perform less than $log_2 n$ iterations when the separator is placed on the searched key. One possible improvement might extend our search algorithms by an additional comparison for equality on each level. Therefore, instead of comparing both SIMD registers only for greater-than relationship, we additionally compare for equality. This additional comparison requires no further load instructions because the search key and the list of keys are already resident in the SIMD registers. However, the additional comparison result must be interpreted using expensive conditional branches with possibly no benefit. A benefit will only emerge, if the search key is equal to a key on an upper k-ary search tree level. In this case, the search may terminates above leaf level and comparisons below this level can be omitted. However, we expect no performance improvements for flat k-ary search trees.

3.5.2 Algorithms for Linearization

Figure 3.9 illustrates our Seg-Tree that rearranges keys inside nodes to enable an inner node search algorithm using SIMD instructions.

We examine two algorithms for linearizing keys in a Seg-Tree node. The first algorithm uses *breadth-first search* while the second algorithm uses *depth-first search* to determine the linearized key order. The *breadth-first*

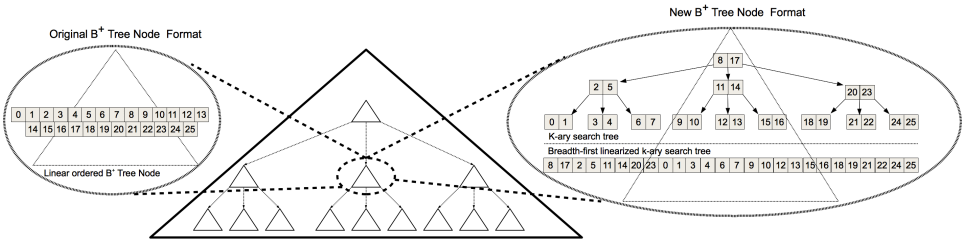


Figure 3.9: B$^+$-Tree node with linear order (left) and breadth-first linearized order (right).

search transformation $P_{BF}(p_L)$ assigns each key in a sorted list of n elements $p_L = (0, \ldots, n-1)$ to a position in the linearized k-ary search tree $(0, \ldots, N-1)$. $N-1$ defines the maximum number of keys. Formula 3.1 calculates the offset recursively on each level of the k-ary search tree. This recursion starts on root level for $R = 0$ with $P_{BF}(p_L, 0)$ and terminates if the last level is reached. The division refers to an integer division without remainder and $S(R) = \lfloor \frac{N}{k^R+1} \rfloor$.

$$P_{BF}(p_L, R) = \begin{cases} \frac{p_L+1}{S(R-1)} + \frac{(p_L+1) \bmod (S(R)k)}{S(R)} - 1, \\ \quad \text{if } (p_L+1) \bmod S(R) = 0, \\ \\ P_{BF}(p_L, R+1) + k^R(k-1) \\ \quad \text{else.} \end{cases} \tag{3.1}$$

The *depth-first search* transformation formula $P_{DF}(p_L)$ is defined in Formula 3.2 and starts with $P_{DF}(p_L) = P_{DF}(p_L, 0)$.

$$P_{DF}(p_L, R) = \begin{cases} \frac{(p_L+1) \bmod S(R-1)}{S(R)} - 1 \\ \quad \text{if } (p_L+1) \bmod S(R) = 0, \\ \\ P_{DF}(p_L, R+1) + (k-1) \\ + \frac{(p_L+1) \bmod S(R-1)}{S(R)}(S(R)-1) \\ \quad \text{else.} \end{cases} \tag{3.2}$$

In general, data manipulations require a reordering of existing keys. In case of an insert operation, a naive approach restores the linear order by sorting the list of keys first, before inserting the new key and linearizing the list again. This naive approach could result in a large reordering overhead. Therefore, the Seg-Tree is advantageous for workloads with few inserts. These workloads benefit from an accelerated search and the reordering overhead can be neglected. For workloads with high insert rates, e. g., OLTP workloads, the reordering overhead probably eliminates the speedup of an accelerated search.

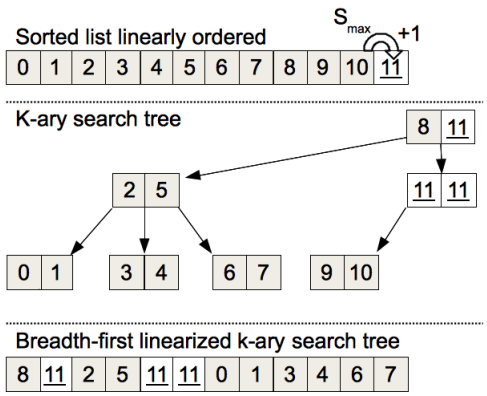


Figure 3.10: Linearization of an incomplete k-ary search tree.

Besides the naive approach, two cases avoid reordering of existing keys. In general, inserting a new key into a linearized node that falls in between two existing keys requires a reordering of all existing keys. However, we can leverage a particular property in case of *continuous filling* with ascending key values. In this case, the inserted key is guaranteed to be greater than all existing keys in the node; thus, the key does not fall in between two existing keys. Therefore, the positions of all existing keys remain unchanged and no reordering is necessary. The new key can be copied directly to its position in the linearized list of keys. The case of *initial filling* is a special case of continuous filling. In this case, a sorted data set will be inserted into an empty Seg-Tree in one batch. Thus, we can leverage this property to speed up tree construction.

Delete operations behave similar to insert operations. Except for a deletion from left to right (increasing values) and from right to left (decreasing values), every random deletion leads to a reordering operation. Update operations always require reordering due to their unpredictable modifications.

3.5.3 Arbitrary Sized Search Spaces

The key count in a B^+-Tree node is bound by the order o of the tree. One node contains at least o and at most $2 * o$ keys. A variable number of keys does not satisfy the requirements of a *perfect k-ary search tree* by Schlegel et al. [SGL09]. A perfect k-ary search tree always contains $k^h - 1$ keys for some integer $h > 0$. A dynamically growing index structure is not capable of satisfying this static property. Therefore, the Seg-Tree must be able to build a k-ary search tree from less than $k^h - 1$ keys. Our SIMD sequence for searching requires only a multiple of $k - 1$ keys. In short, we must extend the k-ary search for an arbitrary number of keys.

Following Schlegel et al. [SGL09], our approach extends the number of keys after linearization if necessary. At first, we identify S_{max} as the largest

available key, i. e., the right most key in a sorted list of keys in ascending order. Next, we transform the sorted list into a linearized order as described in the previous section. Finally, all k-ary search tree nodes with less than $k-1$ keys are replenished with the value of $S_{max}+1$ until each node contains $k-1$ keys. Figure 3.10 illustrates a list of 11 keys. To satisfy the property of $k-1$ keys, we insert $S_{max} = 11$ three times in the k-ary search tree.

Our replenishment approach also affects the search strategy. A search for key v in a k-ary search tree must first check if $v > S_{max}$. If $v > S_{max}$ and the current node is the root or a leaf node, then the search terminates because v does not exist in the Seg-Tree. If $v > S_{max}$ in a branching node, the last pointer at position $n+1$ must be traversed next.

Additionally, appending S_{max} affects the order of a Seg-Tree. In contrast to the original B^+-Tree, the order o of a Seg-Tree specifies no more the minimum and maximum key count in each node. If the combination of k and o does not satisfy the condition of $k-1$ keys, then the maximum and minimum key count in a Seg-Tree node must be multiples of $k-1$. For example, an order $o = 2$ leads to a minimum of two and a maximum of four keys per B^+-Tree node. With $k = 9$, $k-1 = 8$ keys are needed for performing SIMD search. Therefore, a Seg-Tree node must store at least eight keys instead of four keys. Thus, our replenishment approach leads to a larger key count in Seg-Tree nodes if the property of a multiple of $k-1$ keys per node is not satisfied. Our replenishment strategy represents a trade-off between the ability to use SIMD instructions for searching and additional computational effort and memory consumption for storing keys in linearized order. The best node utilization is achieved by storing $k^h - 1$ keys per node. Schlegel et al. [SGL09] suggest another approach for non perfect k-ary trees by defining a *complete tree*.

3.5.4 Seg-Tree Performance

The Seg-Tree performance depends on k-ary search. With larger data type sizes, the k-ary search slows down. Table 3.2 shows common data types and resulting k values for a commonly available 128-bit SIMD bandwidth. For an 8-bit data type and $k = 17$, the k-ary search compares 16 keys in parallel. For a 64-bit data type and the same SIMD bandwidth, the k-ary search compares only two keys in parallel. As a result, the 8-bit data type will perform better. Unfortunately, an 8-bit data type is less likely to be used—usually 32-bit or 64-bit data types are common. In contrast, k-ary search on common data types performs not as good as on small data types. This observation motivated us to develop the *Segment-Trie* to achieve 8-bit k-ary search performance on larger data types.

Table 3.2 illustrates the relationship between common data types and maximal supported k values in a 128-bit SIMD register.

Data type	k value	Parallel comparisons
8-bit	17	16
16-bit	9	8
32-bit	5	4
64-bit	3	2

Table 3.2: k values for a 128-bit SIMD register.

3.6 Segmented Trie

The *Segment-Trie (Seg-Trie for short)* enables the aforementioned performance advantages of k-ary search on small data types for a prefix B-Tree storing larger data types. Following Bayer et al. [BU77] and Boehm et al. [Bea11b], the L bit Seg-Trie is defined on data type D_m with length m bits as:

Definition Segment-Trie: Let Seg-Trie$_L$ be a balanced trie with $r = \frac{m}{L}$ levels $(E_0, \ldots, E_{r-1})$. Level E_0 contains exactly one node representing the root. Each node on each level contains one part of the key with length L (in bits); the so-called *segment*. Each node contains n $(1 \leq n \leq 2^L)$ partial keys. One partial key in one node on level E_i $(0 \leq i \leq r-2)$ points exactly to one node at level E_{i+1}. The nodes on level E_{r-1} contain just as many associated values as partial keys exist. The i-th pointer relates to the i-th partial key and vice versa.

Inserting a key into a Seg-Trie starts by disassembling the key. A key $S[b_{m-1} \ldots b_0]$ is split into r segments $S_0, \ldots, S_{r-1}$ of size L in bits. Each partial key $S_i[b_{L-1} \ldots b_0]$ is composed of $S[b_{(i+1)L-1} \ldots b_{iL}](0 \leq i \leq r-1)$. After disassembling, segments are distributed among different levels $E_0, \ldots, E_{r-1}$. The i-th segment S_i serves as partial key on level E_i.

The search for a key S navigates from the root node on level E_0 to a leaf node on level E_{r-1}. Therefore, S is split into $r = \frac{m}{L}$ segments and each segment will be compared on a different trie level. If a segment does not exist on level E_i, then the search key does not exist in the trie and the search terminates. If the search navigates down to the lowest level and the key exists in the leaf node, then the associated value is returned. Commonly associated values are sets of tuple ids or pointers to other data structures. As a variant of a trie, the major advantage of the Seg-Trie against tree structures is its reduced comparison effort resulting from non-existing key segments. If one key segment does not exist at one level, the traversal ends above leaf level. In contrast, a Seg-Tree will always perform the traversal to leaf level [BM70]. The insert and delete operations are defined similarly.

Suppose an 8-bit Seg-Trie (see Figure 3.11) storing two 64-bit keys $S_i[b_{L-1} \ldots b_0]$ and $K_i[b_{L-1} \ldots b_0]$. A Seg-Trie for a 64-bit data type is capable of storing up to 2^{64} keys. One 64-bit key is divided into eight 8-bit segments

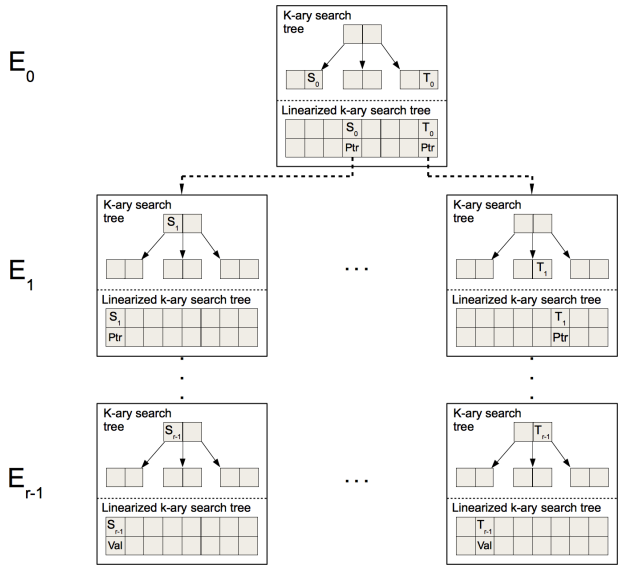


Figure 3.11: Segment-Trie storing two keys.

that are distributed over eight trie levels. Except the root level E_0, each level contains at most 256 nodes and each node points to at most 256 nodes on the next lower level. The nodes on leaf level store the associated value instead of pointers. Each node is able to represent the total domain for the segment data type, e. g., 256 values for 8-bit. Internally, nodes store partial keys in a linearized order. With commonly available 128-bit SIMD bandwidth, the keys inside the nodes are linearized using a 17-ary search tree and 16 keys can be compared in parallel. Each node maintains a k-ary search tree of two levels since $\lceil \log_{17} 256 \rceil = 2$. Therefore, an inner node search for a partial key requires two SIMD comparison operations; one for each k-ary search tree level. For simplicity, the nodes in Figure 3.11 show a k-ary search tree for 8 instead of 256 partial keys. A full traversal of a Seg-Trie with $k = 17$ from the root to the leaves takes at most $\lceil \log_{17} 2^{64} \rceil = 16$ comparison operations. In contrast, a trie using ternary search will perform $\lceil \log_3 2^{64} \rceil = 41$ comparison operations while a binary search trie performs $\lceil \log_2 2^{64} \rceil = 64$ comparison operations for the same number of keys.

Additionally, an 8-bit Seg-Trie leads to an improved cache line utilization. Compared to larger data types, the 8-bit Seg-Trie reduces the number of cache misses due to an increased ratio of keys per cache line. Furthermore, the 8-bit data type offers the largest number of parallel comparison operations. Beyond that, the Seg-Trie offers three additional advantages. First, the corporate prefixes for keys leads to a compression. The Seg-Trie represents a prefix B-Tree on bit level; thus, extending the already existing tries. Second, a fixed number of levels leads to a fixed upper bound for the number of search operations, page, and memory accesses. Third, each level

stores a fixed partition of a key. Therefore, the reorganization following a data manipulation operations is limited to this single node. The remaining trie remains unaffected.

The worst storage utilization for a Seg-Trie occurs when all keys are evenly distributed over the key domain. For example, if the offset between two consecutive keys corresponds to a size such that both keys are stored on different nodes on the same level. Then, all nodes on upper levels are completely filled. However, nodes on lower levels contain only one key. This worst case utilization leads to a poor storage utilization due to sparsely filled nodes. One possible solution to overcome this problem is to swap the assignment of segments and levels. On the other hand, the best storage utilization is achieved when storing consecutive numbers like tuple ids. In this case, the Seg-Trie is evenly filled resulting in a high node utilization.

We identify three cases when no inner node search is necessary: 1) the node is empty, 2) the node contains only one key, and 3) the node is completely filled and contains all possible keys. The first case occurs only for an empty trie. In this case, the search key does not exist in the trie and the search terminates. A node that becomes empty due to deleting all partial keys will be removed. For the second case, if only one key is available in a node, we directly compare this key with the search key without performing a search. In the last case, the node is filled with all possible partial keys of the key domain. Therefore, we directly follow the corresponding pointer for that partial key instead of performing a search. This transforms a node into a hash like structure with a constant-time lookup speed.

Following the idea of *expanding tries* by Boehm et al. [Bea11b] and *lazy expansion* by Leis et al. [LKN13], we suggest to omit tree levels with only one key. Therefore, we create inner nodes only if they are required to distinguish between at least two lower nodes. This approach speeds up the search process and reduces the memory consumption for a Seg-Trie. We refer to this improvement as the *optimized Seg-Trie*. The optimized Seg-Trie stores only levels with at least two distinct keys. Suppose an 8-bit Seg-Trie storing 64-bit keys on eight levels. When filling the tree with consecutive keys starting from 0 to 255, the partial keys are only inserted into one leaf node. After initializing with zero, the seven nodes above leaf level remain unchanged and contain only one partial key throughout the entire key range $[0 \ldots 255]$. Therefore, we suggest to omit the seven levels with only one partial key above leaf level. This reduces the memory consumption and speeds up the trie traversal. When inserting 256, the optimized Seg-Trie increases by one level and creates an additional node on the same level. The optimized Seg-Trie incrementally builds up the Seg-Trie starting from leaf level. To remember the prefixes of omitted level, we store them as an additional information inside the nodes. Other techniques for decreasing the height of a trie by reducing the number of levels are *Bypass Jumper Arrays* suggested by Boehm et al. [Bea11b] and *path compression* suggested by Leis et al.

[LKN13]. Both techniques are also applicable to our Seg-Trie.

In Table 3.3, we contrast Seg-Tree and Seg-Trie by their main differences.

Property	**Seg-Tree**	**Seg-Trie**
Derived From	B-Tree	Prefix B-Tree
Number of Iterations	Tree Height	Max. # Level (Early termination possible)
Number of Level	Dynamic	Static (Pre-defined)
Degree of Parallelism	Depends on key size	Depends on partial key size

Table 3.3: Comparison Seg-Tree vs. Seg-Trie.

3.7 Evaluation

In this section, we experimentally evaluate our tree adaptations for different data types and data set sizes. At first, we describe our experimental setup in Section 3.7.1. We evaluate three algorithms for bitmask evaluation and choose one for the remaining measurements in see Section 3.7.2. Then, we analyze the performance of the k-ary search using performance counters in Section 3.7.3. Finally, we evaluate the performance of our Seg-Tree in Section 3.7.4 and Seg-Trie in Section 3.7.5. The original B^+-Tree serves as the baseline for our performance measurements.

3.7.1 Experimental Setup

All experiments were executed on a machine with an Intel Xeon E5520 processor (4 cores each 2,26 GHz and Intel Hyper Threading). Each core has a 32 KB L1 cache and a 256 KB L2 cache. Furthermore, all cores share an 8 MB L3 cache. The Xeon E5520 is based on Intel's *Nehalem* microarchitecture with a cache line size of 64 byte and a SIMD bandwidth of 128 bit. The machine utilizes 8 GB of main memory with 32 GB/s maximum memory bandwidth. We use the Intel icc compiler with `O2` optimization flag and `SSE4` for SSE support on a Windows 7 64-bit Professional operating system.

We generate a synthetic data set. For 8-bit and 16-bit data types, we generate key sequences for the entire domain of 256 and 65536 possible values, respectively. For 32-bit and 64-bit data types, we generate key sequences containing values in ascending order starting at zero. Initially, we load the entire data set into main memory. After that, we build the tree by creating nodes using the configuration shown in Table 3.4. K results from a SIMD bandwidth of 128-bit and the chosen data type. N_L denotes the number

of keys in the sorted list of keys and N_S denotes the number of keys in the linearized k-ary search tree of height r. N determines the maximum number of keys in one node. The memory consumption of one key consists of a key value and a pointer to the next node level. The size of a pointer on a 64-bit operating system is eight byte and the key size is determined by the chosen data type. To utilize the hardware prefetcher efficiently, we adjust the node size to be smaller than 4 KB. A node size smaller than 4 KB results in no cache miss due to crossing the 4 KB prefetch boundary [Int12b]. Additionally, our node configuration builds a perfect k-ary search tree from k^r keys. Considering the prefetch boundary and perfect k-ary search tree property, we configure the nodes as shown in Table 3.4. The node size is calculated by $N_L+1*sizeof(pointer)+N_S*sizeof(data\ type)$. For example, each node for an 8-bit data type stores $N_L + 1 = 255$ 8-byte pointers and $N_S = 256$ 8-bit keys. We store the keys in one contiguous array. The *cache lines* column expresses how many cache lines are required to access each key in a node. It is calculated by $\frac{N_S*sizeof(data\ type)}{cacheline\ size}$. Using k-ary search, we need one comparison operation on each k-ary search tree level. Therefore, we access at most r cache lines. Notice, that all nodes are completely filled. After building the tree, we measure the time for searching x keys in random order and calculate the average search run-time for one search operation. For the remainder of this chapter, we define $x = 10,000$. To measure the run-time we use RDTSC (*Read time-stamp counter*) instructions to count the clock cycles between two points in time. All measurements are performed in a single thread. There is no output written to disk and the search result is not further processed.

Data type	**k**	**N$_L$**	**N$_S$**	**r**	**N**	**Node size**	**Cache lines**
8-bit	17	254	256	2	289	2296	2
16-bit	9	404	408	3	729	4056	7
32-bit	5	338	344	4	625	4096	11
64-bit	3	242	242	5	243	3880	16

Table 3.4: Node characteristics.

3.7.2 Bitmask Evaluation

As described in Section 3.3.1, our SIMD sequence compares two SIMD registers and outputs the result into a third SIMD register. The resulting bitmask in the third SIMD register must be evaluated to determine the relationship between the search key and the list of keys. For bitmask evaluation, we analyze three algorithms which we introduce in Section 3.3.2, i. e., *bit shifting*, *switch case*, and *popcount*. At first, all algorithms use the `movemask` instruction to create a 16-bit bitmask from the most significant bits in the result

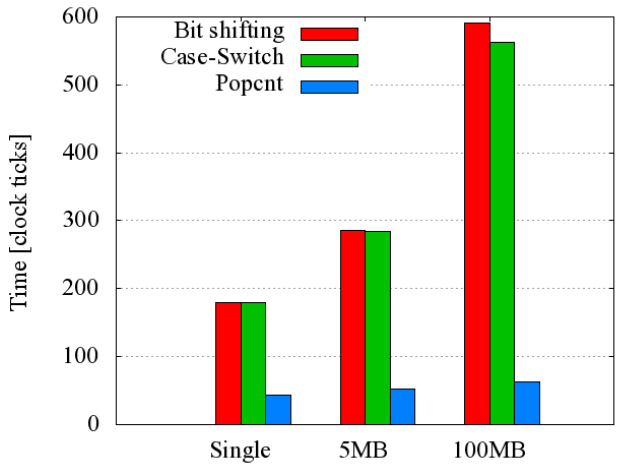


Figure 3.12: Evaluation of bitmask for 8-bit data type.

SIMD register and place the bitmask into the lower 16 bits of an x86 register. The algorithms differ in converting the 16-bit bitmask into a position in a sorted list of keys. Figure 3.12 shows the results for the three algorithms performing a search in an 8-bit Seg-Tree. The three categories *Single*, *5 MB* and *100 MB* represent the amount of data in the Seg-Tree. For the remainder of this evaluation, we refer to *Single* as a data set containing keys in one single node. With *5 MB* and *100 MB*, we refer to upper bounds for the data set size. The resulting node count depends on the single node size and the upper bound (see Table 3.4).

As shown in Figure 3.12, the *popcount* algorithm achieves the best overall results and is also independent of data set size. The main reason for its superiority is the elimination of 16 conditional branches; thus, eliminating expensive pipeline flushes. Thus, performance improvements of k-ary search originates mainly from eliminating conditional branches. For larger data types, there are less conditional branches available which can be eliminated. Therefore, the decreasing number of conditional branches for larger data types leads to a decrease in k-ary search performance. The largest data type provided by Intel [Int12b], i. e., 64-bit, performs only two conditional branches. Due to the overall best performance, we use the *popcount* algorithm for the following evaluation of our Seg-Tree and Seg-Trie implementation.

3.7.3 Evaluation K-ary Search

In this section, we compare the k-ary search on depth-first and breath first data layout against the common binary search. We build a perfect k-ary search trees of different levels and compare their utilization of the memory

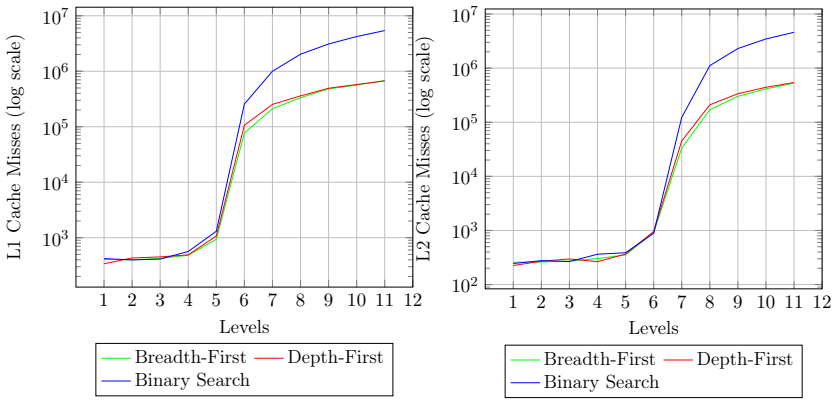
LVL	TupleCnt	size in KB	Cache-Mem
1	5	0.02	
2	25	0.10	
3	125	0.49	
4	625	2.44	
5	3125	12	fit L1
6	15625	61	fit L2
7	78125	305	
8	390625	1,526	
9	1953125	7,629	fit L3
10	9765625	38,147	
11	48828125	190,735	
12	244140625	953,674	

Table 3.5: Test Configuration.

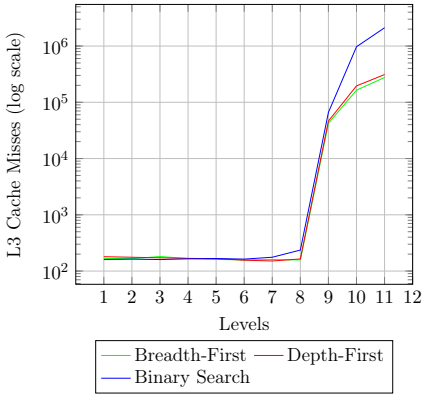
hierarchy. For this test, we perform 100K random key searches on 32-bit keys. For a 128-bit SIMD register, a key size of 32-bit leads to $k = 4$; thus, four keys are loaded into one SIMD register. In Table 3.5, we show the resulting number of entries, the index size, and an information in which cache level an index of this size fits. In Figure 3.13, we show cache misses in the L1, L2, and L3 cache. On the x-axis, we plot the number of levels of the k-ary tree. For small tree sizes that fit into a particular cache level, the number of cache misses do not differ between k-ary search and binary search. However, as soon as the tree exceeds the cache size, the binary search induces up to a factor of ten more cache misses compared to k-ary search. As shown in Table 3.5, a tree exceeds L1 cache size starting from a level count of 6, L2 cache size from a level count of 7, and L3 cache size from a level count of 9. Comparing both k-ary searches, the breadth-first layout leads to slightly less cache misses compared to the depth-first layout. The main reason for the superior cache behavior of the k-ary search is their cache line utilization. K-ary search fully utilizes each cache line. In contrast, binary search exploits in general only one data time per cache line. Thus, k-ary search reduces the number of loaded cache lines significantly and thus improves performance.

3.7.4 Evaluation Seg-Tree

We evaluate the Seg-Tree using four different integer data types (8-, 16-, 32-, and 64-bit) as keys and store three differently sized data sets (Single, 5 MB, 100 MB). Figure 3.14 shows the average run-time of one search operation in clock ticks using different inner node search algorithms. The red bar presents the original B^+-Tree using binary search. The Seg-Tree uses SIMD search on breadth-first (green bar) and depth-first (blue bar) linearized keys.



(a) L1 Cache Misses. (b) L2 Cache Misses.



(c) L3 Cache Misses.

Figure 3.13: Breadth-First vs. Depth-First Search.

The measurements show, that the depth-first search performs best in all configurations. Generally, the performance increases for smaller data types. This observation is independent of data set size and can be explained by two reasons. At first, for 8-bit data type values, 16 comparison operations can be performed in parallel while for 64-bit data type values, only two are possible. Second, small data type values lead to a better cache line utilization due to an increased ratio of keys per cache line. The k-ary search on 8-bit data type values outperforms the binary search nearly by a factor of eight even for large data set sizes.

For large data set sizes, the SIMD search performance on breadth-first and depth-first linearized keys is nearly similar, except for an 8-bit data type. For decreasing data set sizes, a Seg-Tree using depth-first linearized keys outperforms a Seg-Tree using breadth-first linearized keys. The cache hierarchy impacts the performance of both Seg-Trees and the B^+-Tree. For a single node, the node resides most likely in the L1 cache for each search operation. Therefore, the *Single* category illustrates the pure run-time for each search

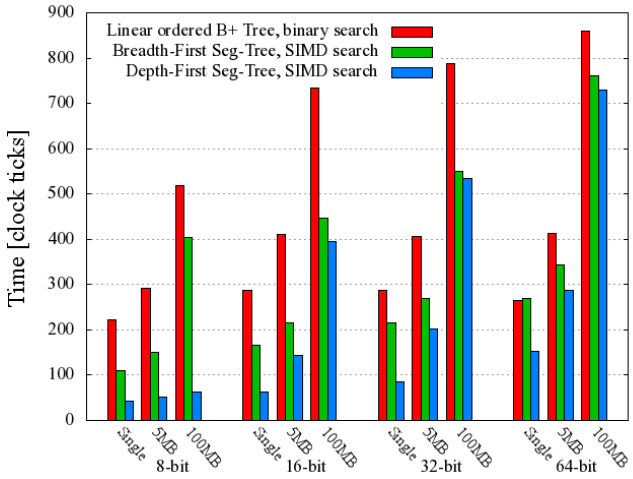


Figure 3.14: Evaluation of Seg-Tree.

algorithm in a comparable way by excluding cache effects. For a 5 MB data set size, the entire data set will properly fits into the 8 MB L3 cache but not entirely in the 256 KB L2 cache. Thus, an access to a random node has a possibility to produce a L2 cache miss. The 100 MB data set fits in no cache level entirely; thus, further increases the impact of cache misses. The computational effort for searching inside the nodes become more negligible with an increasing number of cache misses. The cache hierarchy becomes the bottleneck for larger data set sizes. Generally, the inner node search algorithms transform from a computation bound algorithm to a cache/memory bound algorithm for increasing data set sizes.

3.7.5 Evaluation Seg-Trie

We evaluate the Seg-Trie and optimized Seg-Trie against different Seg-Trees in Figure 3.15. The speedup refers to the original B^+-Tree using binary search. The optimized Seg-Trie implements the elimination of levels as mentioned in Section 3.6. The node configuration for the Seg-Tree is equal to the 64-bit data type configuration in Table 3.4. The Seg-Trie contains always eight levels and the optimized Seg-Trie contains at most eight levels. Each trie node follow the 8-bit data type configuration in Table 3.4. The depth of the tree in Figure 3.15 refers to the number of levels that are filled with keys. We vary the number of keys to fill the expected level count. For comparability reasons, all tree variants contain the same number of levels and keys. To achieve this, we skew the data for both Seg-Trie variants to produce the expected level count.

As shown in Figure 3.15, the performance of a Seg-Trie increases almost linearly with the depth of the tree. The performance is measured against a

B^+-Tree using binary search. Instead of comparing a 64-bit search key with a 64-bit key on each level like the B^+-Tree using binary search, the Seg-Trie compares only one 8-bit part of the search key on each level. Additionally, an increase of tree depth by one for a Seg-Trie leads to no additional node comparison because a 64-bit Seg-Trie always searches among eight tree level. In contrast, the B^+-Tree using binary search must perform one additional node search for each additional tree level. Therefore, with increasing tree depth, the speedup of the Seg-Trie compared to the B^+-Tree using binary search increases almost linear.

The optimized Seg-Trie provides a constant speedup independent of tree depth. As mentioned in Section 3.6, the optimized Seg-Trie omits levels with less than two distinct values. The depth of the tree in Figure 3.15 refers to the number of filled levels. Compared to a Seg-Trie, the optimized Seg-Trie requires one node comparison on each filled tree level. In contrast, a Seg-Trie always performs eight comparisons even for levels containing only one key. Thus, the number of node comparisons for the optimized Seg-Trie increases for deeper trees. The speedup is constant because it is measured against the B^+-Tree using binary search. Each additional tree level adds one additional node to both tree variants. Therefore, the speedup remains unchanged. Suppose, we insert a 100 MB data set containing nearly 1.6 M keys in consecutive order (starting at zero) into a 64-bit optimized Seg-Trie. We need 21 bits out of the available 64 bits to represent the largest key representation (1,638,400). Thus, the upper 43 bit are unused. The number of levels that can be omitted due to 43 unused bits depend on the size of the partial keys. In our example, we split a 64-bit key into eight parts. Therefore, the optimized Seg-Trie of depth three omits five out of eight levels, i. e., 40 bits. For a tree depth of eight, no levels are omitted and both Seg-Trie variants behave similar. The reduced number of levels leads to a reduced amount of memory transfers, a reduced possibility of cache misses, and less computational effort.

The Seg-Tree using breadth-first linearization provides a constant speedup of 113% compared to a B^+-Tree using binary search and is independent of tree depth. The Seg-Tree using depth-first linearization provides an 118% improvement with same characteristics. Therefore, both lines overlap in Figure 3.15. Like the B^+-Tree using binary search, the Seg-Tree adds one node to the traversal path for each increase in tree depth. Therefore, the speedup remains constant.

The smallest data type that can currently be processed by the *SIMD Extensions* is 8-bit [Int12b]. This restriction limits a further increase in tree depth. However, the optimized Seg-Trie and the Seg-Tree are independent of tree depth. The Seg-Tree performs poorly on large data types but increases its performance for smaller data types. The optimized Seg-Trie provides a constant 14 fold speedup independently of tree depth and an eight fold reduced memory consumption compared to the original B^+-Tree.

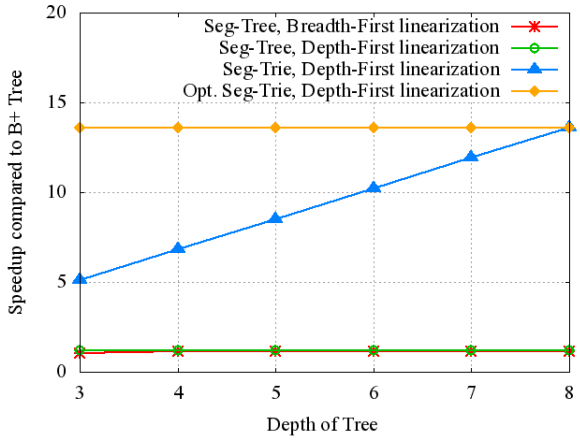


Figure 3.15: Evaluation Seg-Tree vs. Seg-Trie for 64-bit key.

3.8 Summary

This chapter introduces the Seg-Tree and Seg-Trie which enable efficient SIMD usage for tree and trie structures. We showed that SIMD instructions of modern processors are qualified to speedup tree-based index structures. Therefore, we make SIMD instructions applicable for tree based search algorithms in modern database systems. Based on k-ary search by Schlegel et al. [SGL09], we investigate how to use this approach for a B^+-Tree and prefix B-Tree structure. We contribute two different linearization and search algorithms, the generalization to an arbitrary key count, and three algorithms for bitmask evaluation. The introduced Seg-Trie takes advantages of k-ary search for small data types and enables them for large data types. Furthermore, our optimized Seg-Trie provides a 14 fold speedup and an 8 fold reduced memory consumption compared to the original B^+-Tree. We emphasize, that the strength of a Seg-Trie arises from storing consecutive keys like tuple ids. On the other hand, if keys are evenly distributed, the Seg-Trie needs further adjustments to enhance the storage utilization. As the SIMD bandwidth will increase in the future [AVX08], index structures using SIMD instructions will further benefit by increased performance.

The research work presented in this chapter could be extended in two areas. First, this work focuses on optimizing the Seg-Tree and Seg-Trie for single thread performance. Future work could investigate the impact of multi-threading, multi-core, and many-core architectures on different aspects of Seg-Tree and Seg-Trie processing. Especially, the impact of SIMD instructions on concurrently used index structures could be an interesting research topic. Second, future work could adapt the Seg-Trie and Seg-Tree for GPU processing. In this area, the suitability of GPU supported scatter and gather operations could be examined.

Chapter 4

Scheduling Query Execution

Over the last decades, the clock speed per core reached a plateau due to physical limitations. Since then, an increasing number of available on-chip transistors are used to incorporate more processors and larger cache. Additionally, a large amount of commonly available main memory allows modern database systems to store their entire working sets in main memory. However, memory access latency and memory bandwidth between main memory and CPU improved differently. Nowadays, CPUs process data much faster than transferring data from main memory into caches. This trend creates a *Memory Wall* which is the main challenge for modern main memory database systems [Aea99, Bea99].

Research in the last decade also shows, that parallelization and chip multiprocessing exacerbate this *Memory Wall* [Aea99, Bea99]. The ever increasing number of processing units per chip have to share a constant memory bandwidth which reduces the available memory bandwidth per processing unit. An uncoordinated parallel access to shared data structures from different processing units leads to a *memory bottleneck* [Bea99]. To overcome the memory bottleneck, the locality of data and instructions become increasingly important. The cache hierarchy of modern processors alleviates the memory bottleneck by reusing already loaded data and instructions in caches. The reuse of data is exploited by either accessing data that was already loaded before, i. e., *temporal locality*, or, by accessing data that is located contiguous to already loaded data, i. e., *spatial* locality.

Caches in modern CPUs reduce the gap between main memory and CPUs by caching frequently used data and instructions. However, caches cannot be controlled directly. Thus, a database might only guide cache behavior by indirect means like data placement and access patterns. The exploitation of these indirect means are vital for chip multiprocessors to supply each CPU with sufficient data despite limited main memory bandwidth.

Due to the inherent parallelism of database systems, the opportunities of chip multiprocessors are in particular applicable. On the other hand, a

DBMS may also exhibit tight data dependencies that require some degree of synchronization between operators executed in parallel. To exploit parallelism in databases, different approaches for parallel query execution have emerged over the last decade [CRG07, Pea90, Rea13, Bea83, BLP11]. The performance of these approaches is mainly affected by the non-manageable cache hierarchy. However, each approach exploits the capabilities of modern processors differently. Furthermore, the comparison is difficult due to various operator-to-resource assignments during run-time (scheduling strategy) and the number of tuples each operator processes (chunk size).

In this chapter, we first classify common DBMS by their scheduling strategies and chunk sizes. Then, we propose a task model called *Query Task Model (QTM)* that opens a design space for database schedules. QTM allows us to express and compare different approaches for parallel query execution. With QTM, we generalize the modeling of parallel query execution such that different approaches become comparable. Using QTM, we model an arbitrary QEP as a set of tasks. Each task represents a particular piece of work on a subset of data.

Our evaluation of different schedules modeled in QTM shows, that a tuple-at-a-time schedule cannot exploit modern hardware efficiently. In contrast, an operator-at-time schedule increases the performance due to increased cache utilization. However, a buffer-at-a-time schedule that takes the cache hierarchy into account outperforms schedules that do not. Furthermore, we will show that a schedule that is optimized for data cache locality does not necessarily outperform a schedule that is optimized for instruction cache locality. We identify a sweet spot where the ratio of data locality and instruction locality produces the fastest schedules. In the light of this discussion, our contributions are as follows:

- We classify common DBMS by their scheduling strategies and chunk sizes.
- We define QTM (Query Task Model), a model that allows to express and compare different approaches for parallel query execution.
- Using QTM, we compare common query execution schedules regarding their resource utilization.
- Based on our analysis, we identify a sweet spot that produces the fastest schedules.

The remainder of this chapter is organized as follows. In Section 4.1, we classify common DBMS by their scheduling strategies and chunk sizes. Next, we contrast our model to current state-of-the-art scheduling strategies. In Section 4.2, we introduce our *Query Task Model (QTM)*. Then, we model common database schedules with QTM in Section 4.3. After that, we present our evaluation results in Section 4.4. Finally, we conclude and outline future

work in Section 4.5. Our QTM model presented in this chapter was published in [ZF14].

4.1 Scheduling in Databases

In this section, we classify common database schedulers by their chunk size and scheduling strategy. After that, we contrast our approach for modeling query execution using tasks to common state-of-the-art schedulers.

4.1.1 Classification of Database Schedulers

Over the last decades, different approaches for parallel query execution have emerged because scheduling a database query exhibits some degrees of freedom. In the following, we present four alternatives a database scheduler might exploit when optimizing query execution with respect to available resources.

First, a scheduler has to determine an execution order for available operators among queries. For a single query, the execution order has to satisfy the constraints introduced by a QEP. For multiple queries, the execution order for pending queries has to take fairness and priorities into account. Second, a scheduler has to assign a degree of parallelism (DOP) to each operator. A DOP can be either determined statically during compile-time or dynamically during run-time. Third, a scheduler has to specify a degree of thread cooperation. In general, threads can either work cooperatively on the same operator or separately on different operators. Finally, a scheduler has to partition the input for each operator. The size of a partition, so-called *chunk size*, determines how many tuples are processed by an operator before returning the result.

In this chapter, we focus on operator scheduling for a single query during run-time. Furthermore, we assume an invariable QEP that was generated by a query optimizer as an input. In the following, we classify different database schedulers by their scheduling strategies and chunk sizes as shown in Figure 4.1. The scheduling strategy controls the processing of different operators by different processors. The chunk size determines the number of tuples processed by each operator instance and ranges from one tuple, over multiple tuples (N), to an entire column. At first, we show how early databases implement scheduling before introducing three classes of state-of-the-art database schedulers in Section 4.1.1.1. Then, we describe proposed chunk sizes and their impact on query execution in Section 4.1.1.2. Finally, we show possibilities for exploiting parallelism in databases as well as different degrees of thread cooperation in Section 4.1.1.3.

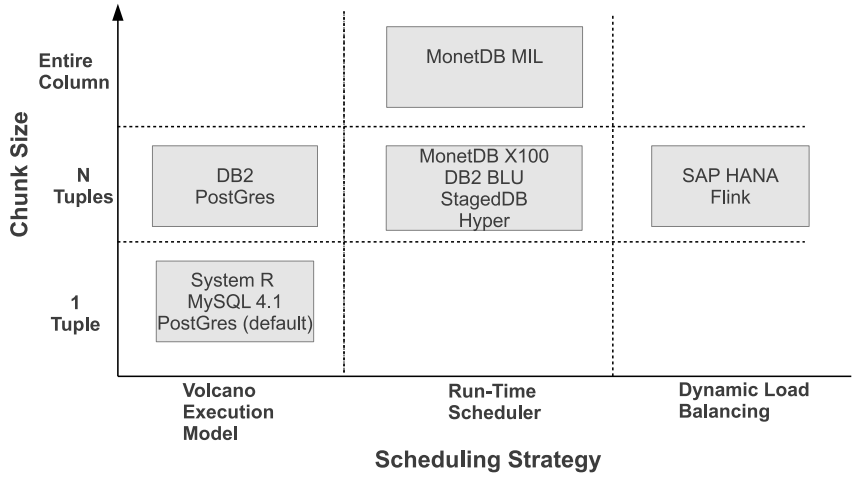


Figure 4.1: Classification of Databases Schedulers.

4.1.1.1 Scheduling Strategies

The first dimension for our DBMS classification is based on the applied *scheduling strategy* (see Figure 4.1). Early DBMSs determined the DOP of an operator statically based on resource availability at compile-time [Pea90, Zea93]. Thus, they exploit a *static optimization approach*. During run-time, the DOP was implemented by a static assignment of threads to operators. The main disadvantage of this approach is the temporal gap between compile-time and run-time. During run-time, the system load might be quite different which may lead to a suboptimal resource utilization. Furthermore, uncertain information at compile-time such as wrong cardinality estimates, skewed data, correlated attributes, outdated statistics, or user-defined functions, may also lead to a suboptimal decision [KD98]. The result of these uncertainties could be a wrong prediction of operator work (execution skew) that leads to an imbalanced work distribution. Additionally, a static assignment of threads to operators introduces a *discretization error*. Since operators and processors are discrete entities, a fixed number of operators cannot be assigned to a fixed number of processors such that each operator reaches its optimal DOP [Bea96]. Finally, a static assignment may lead to a *pipeline delay problem*. Therefore, processors that are assigned to operators at the end of an execution plan idle at the beginning and processors at the beginning idle at the end of the query execution [MOW97]. To response to these compile-time uncertainties, three different classes of database schedulers have emerged.

A first class of state-of-the-art database schedulers responds to compile-time uncertainties at run-time. For example, XPRS implements a *two-phase optimization approach* [Pea90, Hon92]. In the first phase, the optimizer ignores aspects of parallelism and produces the best sequential plan during

compile-time. In the second phase, the plan is optimized for parallelism during run-time. Therefore, a plan is decomposed into a set of plan fragments and the applied DOP is determined based on the current resource availability. After decomposition, a *parallel executor* determines a processing schedule and distributes fragments for execution. This approach leads to an improved resource utilization because it takes resource availability at run-time into account to determine DOP. Following this approach, *dynamic optimization* during run-time becomes a vital source for query execution performance and thus modern database systems implement a dedicated *run-time scheduler*. A run-time scheduler is able to react to changes in the system load or incorrect estimations during run-time. Over the last decades, even more complex scheduler have been proposed [Bea05, Rea13, Lea14, HA05]. For instance, one run-time scheduler takes NUMA-characteristics into account [Lea14] and another approach implements a time-slice based scheduling algorithm [HA05].

A second class of state-of-the-art database schedulers responds to compile-time uncertainties with *dynamic load balancing*. At compile-time, a query plan is disassembled into tasks that are placed into a queue. During run-time, each processor dequeues tasks until the queue is entirely processed. A dynamic load balancing approach omits a dedicated run-time scheduler because the actual mapping of threads to operators is implemented using a *work-pull* strategy. Thus, each processor dynamically acquires new work on its own if computing capacities are available. The execution order of operators is determined by the order of tasks in the queue. The DOP of an operator is not statically defined and depends on the number tasks currently executing this operator. Furthermore, predicting which processor executes which task reveals high uncertainties because differently sized tasks and varying resource availability introduce high variability. The proposed approaches in research vary in the number of queues and the granularity of tasks [Bea96, MOW97, LT92, Pea13].

A third class of state-of-the-art database schedulers responds to compile-time uncertainties with a simple execution model. The demand-driven *Volcano execution model* emerged as the most commonly used scheduling strategy [Gra90, ZR04, Neu11]. This model hides aspects of parallelism from operators and omits a dedicated run-time scheduler. Instead, the Volcano execution model implements an *open-next-close* iterator interface for each operator. The *open* call initialize an operator and the *close* call deallocates all resources. The *next* call on one operator propagates recursively to its child operators until one output tuple is generated. Through repeated *next* calls, all tuples are processed and the operator can be closed. The actual assignment of resources to operators is implicitly implemented by this model. Thus, this model makes parallel query execution entirely self-scheduling [Gra90]. The advantages of the Volcano model are the avoidance of synchronization and scheduling, minimized data copies, reuse of current data items in main

memory, and lazy operator evaluation [ZR04]. Graefe extends this model for parallel execution by introducing *exchange* operators to synchronize different threads executing the same query plan [Gra90].

4.1.1.2 Chunk Size

The second dimension for our DBMS classification is based on the *chunk size* (see Figure 4.1). The input of an operator can be divided into multiple chunks (partitions) which can be processed in parallel by different operator instances (intra-operator parallelism). The chunk size determines how many tuples are processed by an operator instance before the result is returned. As a result, the chunk size determines the number of operator calls.

The chunk size is defined either at compile-time, run-time, or as a constant value. However, all DBMS shown in 4.1 define a fix chunk size for all QEPs. An alternative approach proposed by Cieslewicz et al. [Cea09] suggests to change the chunk size dynamically based on cache miss sampling during run-time.

The chunk size in common DBMSs varies significantly. Some approaches define a chunk size in relation to a hardware parameter [Zea08, Pea01, CRG07, ZR04]. However, most approaches state, that they adjust the chunk size such that the entire chunk fits into a certain cache [Bea05, Rea13]. Thus, common chunk sizes match L1, L2, or L3 cache sizes. Other approaches define a fix number of tuples [Lea14] or a fix block size like 64 KB [Sea05]. Two extremes are a chunk size of one tuple used in the classical Volcano execution model [Gra90] and a chunk size of one column used in MonetDB/MIL [Bea99].

Block-oriented processing [Pea01] extends the Volcano execution model by changing the number of tuples transferred between two operators from one tuple to a *block* or a chunk of tuples. Thus, by grouping tuples into chunks, the data locality is improved and the overhead of one operator call is amortized over multiple tuples. In general, block-oriented processing increases the performance as long as the entire block of tuples fits into a cache [Zea08]. Zhou and Ross [ZR04] implement block-oriented processing by inserting buffers between certain operators which shows an improved instruction cache performance. Zukowski et al. [Zea08] compare the row-wise storage layout (NSM) and column-wise storage layout (DSM) in combination with block-oriented processing. They show, that the storage layout strongly impacts the performance of different database operations.

4.1.1.3 Degree of Parallelism

The degree of parallelism is orthogonal to the dimension shown in 4.1. Therefore, a degree of parallelism has to be specified in any scheduling strategy. However, to specify the DOP of an operator, the operator type has to be

taken into account. Database operators can be classified into *blocking* and *non-blocking* operators. A *blocking* operator, e. g., sort or aggregation, needs to collect all input tuples before it produces the first output tuple. In contrast, a *non-blocking* operator, e. g., selection or hash probe, produces output tuples on-the-fly. A sequence of non-blocking operators in a producer/consumer relationship represents a so-called *pipeline*.

Blocking operators are parallelized by exploiting *intra-operator* parallelism. Thus, the input of a blocking operator is partitioned into chunks and might be processed in parallel. A scheduler implements intra-operator parallelism by disassembling one operator into multiple operator instances. During run-time, each operator instance processes one chunk. In general, the number of instances is determined by $\lceil \frac{number\ of\ input\ tuples}{chunk\ size} \rceil$. In contrast, a pipeline containing only non-blocking operators enables a much higher flexibility for parallel processing. A scheduler might parallelize a pipeline by additionally exploiting *inter-operator* parallelism. Inter-operator parallelism enables parallel processing of different operators without blocking. Finally, *independent parallelism* can be exploited if two pipelines exhibit no dependency. In this case, both pipelines may be processed in parallel [GI96].

Finally, the degree of parallelism has to take the degree of thread cooperation into account. In general, threads can be either working cooperatively together on the same operator instance or each thread works on its own operator instance. Cieslewicz et al. [CRG07] propose a parallel buffer that is filled by a group of threads cooperatively before the buffer is delivered to the next operator. Thus, all threads work on the same operator instance. An alternative approach proposed by Cieslewicz et al. [Cea09] implements a strategy where a distinct chunk of tuples is assigned to each thread. Thus, each threads processes one operator instance independently.

4.1.2 Scheduling Approaches in Database Systems

In this chapter we will propose a dynamic load balancing approach using tasks to model different database schedules. In this section, we contrast our model for query execution against common state-of-the-art scheduling approaches. Therefore, we present existing approaches for query execution and highlight their shortcomings.

Approaches for dynamic load balancing in research vary in the number of task queues and the granularity of tasks [Bea96, MOW97, LT92, Pea13]. There are approaches using one global task queue [MOW97], one queue per thread per operator [Bea96], one queue per processor and one global queue [LT92], or one queue per processor socket [Pea13]. While a global queue constitutes one single point of synchronization, different queues exhibit the risk that one queue becomes empty while other queues still contain tasks. In this case, some kind of work-stealing mechanisms must be established.

The granularity of tasks also vary among different approaches. While one approach did not state how to convert a QEP into a set of tasks [Bea96], two approaches create tasks mainly from partitionable operators like aggregations or hash builds [LT92, Pea13]. Manegold et al. [MOW97] use the call of one operator with one tuple as the basic granularity of one task. However, neither of these approaches considers a task granularity different from one operator call for one tuple. Additionally, locality of data and instructions inside a cache hierarchy and different execution orders are not considered. In contrast, we extend the notion of tasks by a generalized work and data specification and a declaration of processing strategies which specify task execution during run-time.

The optimal chunk size was only examined for a particular scheduling strategy. Padmanabhan et al. [PMJA01] introduce *block-oriented* processing that extends the volcano query execution model to process a block of tuples. Zhou and Ross [ZR04] implement the block-oriented approach by inserting buffers between certain operators to improve the instruction cache performance. Zukowski et al. [Zea08] compare the row-wise storage format (NSM) and column-wise storage format (DSM) in combination with the block-oriented approach. Depending on the storage format, the buffer contains tuples either entirely (NSM) or only one attribute of each tuple (DSM). However, neither of these approaches considers the impact of different scheduling strategies nor take the exploitation of pipeline parallelism into account. In our model for query execution, we take different scheduling strategies and pipeline parallelism into account.

Previous work sampled commercial DBMS workloads to identify the distribution between time spent for computation and time spent for waiting for data. In our context, the OLAP workloads in [Bea05, Hea07c, Aea99] are most relevant. Ailamaki et al. [Aea99] examined four major commercial database systems for their performance on different hardware architectures. They exploit *clocks-per instruction (CPI)* as a metric when executing benchmarks. Even for simple database queries, CPI values are rather high. This indicates, that database systems are particularly ineffective in taking advantage of modern superscalar processor capabilities [Aea99, Bea05]. Furthermore, Ailamaki et al. [Aea99] discovered that on average, half of the execution time is spent in stalls while 90% of the memory stalls are due to L2 data cache misses and L1 instruction cache misses. Other research show similar time and stall distributions [Kea98, Rea98]. Tözün et al. [TGA13] point out, that for OLTP workloads, the L1 instruction cache misses have deeper impact than data cache misses. However, most of the studies use CPUs without now commonly available L3 cache [Aea99, Hea07c, Kea98, Rea98]. Therefore, the L2 data cache stall time will probably move to L3 data cache stall time. In general, they show that databases are particularly ineffective in taking advantage of modern superscalar processor capabilities [Aea99, Bea05]. Our evaluation of common database schedules will contribute to this obser-

vation by adding the chunk size and scheduling strategy as new dimensions that impact the distribution of cache misses.

Another approach for exploiting locality and capabilities of modern hardware is the *StageDB* [HA03, HA05]. StageDB tries to convert the *work-centric* execution model into a *data-centric* execution model; thus, creating a staged, data-centric DBMS design. StageDB breaks the query plan down into *self-contained* stages. Each stage represents one operator in the query plan. The stages exchange data through packages that are routed through different stages, according to the query plan. The packages carry their state and private data across stages. Each stage owns one input queue for arriving packages and applies its operation to each package in its queue. After that, packages are routed to the next stage by placing packages into the input queue of the next stage; thus, establishing a *push-based* communication model. The parallelization and synchronization is handled by one global scheduler for the entire DBMS and a local scheduler for each stage. The global scheduler assigns a time quantum to each stage. Within each stage, the local scheduler assigns threads to work. The local scheduler follows the idea of autonomous stages with explicit data and instruction locality and minimized synchronization to other stages. The global scheduler reassigns control to another stage if 1) the time quantum of one stage is exhausted, or 2) the input queue is empty, or 3) the output queue is full. Therefore, the global scheduler is mainly responsible for load balancing. In contrast to other approaches, StageDB does not only try to minimize uncoordinated memory access of different threads, but also minimizes the uncoordinated memory access from different queries. It represents a form of *multi-query* optimization. If two queries use the same operator in a proximity of time, both queries place the same work in the same stage at the same time. As a result, the work has to be performed only ones. If enough similar queries are submitted in parallel, this might reduce the computational effort as well as data transfers from and to memory.

With *QPipe*[HSA05], the work sharing across concurrent queries is further increased by introducing *on-demand simultaneous pipelining (OSP)*. With OSP, the output of one stage can be simultaneously pipelined to multiple other stages. On example that benefits from OSP is the concurrent scan of one table from multiple queries. To enable OSP, QPipe introduces an *operator-centric relational engine*. The common DBMS design of *one-query, many-operators* is moved to *one-operator, many-queries*. Psaroudakis et al. [PAA13] extend the sharing opportunities among concurrent queries in QPipe by combining *Simultaneous Pipelining*, that shares intermediate results of common sub-plans, and the approach of *Global Query Plans*, that introduces *shared operators* to share common work. However, Johnson et al. [Jea07] and Psaroudakis et al. [PAA13] pointed out, that work sharing across concurrent queries are not always beneficial. There is a trade-off between exploiting work sharing opportunities and the available parallelism. Especially

the producer/consumer pattern with one producer and multiple consumer might introduce a serialization point along the critical path of the data flow. Psaroudakis et al. [PAA13] suggest to change the push-based communication model into a pull-based communication model to encounter this problem.

Another application of the *StageDB* approach is *STEPS* [HA04]. STEPS improves the instruction cache performance for transactional workloads by multiplexing concurrent transactions and exploiting common code paths. Besides the work sharing opportunities of the StageDB approach, there are also some challenges for parallelization. First, there is the need for a global scheduler that introduces explicit synchronization between the stages. Second, the routing of tuples through different stages introduces some additional overhead for bookkeeping. Third, the global scheduling schema does not take the temporal locality of tuples between stages into account. If StageDB executes only one query, the scheduler assigns the time slices in a *step-wise manner*. Therefore, similar to the block oriented approach, the global scheduler processes a number of tuples (similar to a buffer) sequentially by each stage. After that, the scheduler starts a new round with the next tuples. This pattern supports temporal locality but concurrent execution of different query plans with different operator sequences may lead to cache thrashing when switching between stages of different queries. Furthermore, the back pressure of differently paced producer/consumer reduces temporal locality too. Finally, StageDB requires a extensive change of the common query execution engine. In contrast to this approaches for multi-query optimization, we focus in this chapter on the single-query execution and its optimization.

4.2 Query Task Model

In this section, we present our *Query Task Model* (QTM) that opens a design space for database schedules. With QTM, we generalize the modeling of parallel query execution such that different approaches become comparable.

Section 4.2.1 introduces QTM as a task model for query execution before we describe the implementation of QTM in Section 4.2.2. After that, we define QTM formally in Section 4.2.3. Finally, we describe different aspects of parallelism in QTM in Section 4.2.4.

4.2.1 QTM Overview

The task paradigm is a common abstraction for parallelism in high performance computing environments [MSM04]. A general task model for parallel computing consists of *tasks*, *units of execution* (UEs), and *processing elements* (PEs). A *task* corresponds to a certain part of an algorithm and is implemented by grouping a sequence of instructions. During execution, each task is mapped to an *unit of execution* (UE) that is either a thread or a process. An UE has to be executed by a *processing element* (PE). A PE is

a generic term for a hardware unit that is either a processor or a machine [MSM04].

We extend this general task model to model parallel query execution of database queries (QTM). This extension allows us to express and compare different approaches of query execution. The main challenge is to include database specific constraints and requirements. To execute a database query using a task model, we have to create tasks from a query execution plan (QEP), map tasks to UEs, and schedule UEs to PEs for execution. In QTM, we create tasks by disassembling a QEP at compile-time. Figure 4.2 illustrates our three-step query transformation process to transform a QEP into a set of tasks that is modeled in QTM.

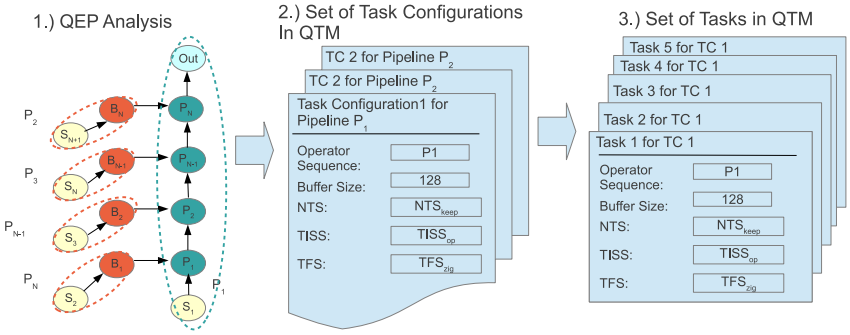


Figure 4.2: QEP Transformation Process.

The transformation process as presented in Figure 4.2 proceeds as follows. In a first step, we analyze a QEP to identify a set of pipeline fragments with maximal length and generate a dependency graph describing their relationships. The dependency graph reveals ordering constraints between operators. In a second step, we group operators into *task configurations* (TCs). Each TC represents a particular piece of work of a QEP on a subset of data. Based on TCs, we describe dependencies and potential concurrent execution for a group of operators. A TC represents a blueprint that specifies the work (operator sequence) and data (buffer size) for its tasks (see Section 4.2.3.1). Furthermore, TCs define three *processing strategies* for task execution during run-time (see Section 4.2.3.2). The mapping of operators to TCs exhibits some degree of freedom. Possible mappings range from a *fine-grained* mapping of one operator to one TC up to a *coarse-grained* mapping of an entire pipeline to one TC. Note, TCs are only used as an intermediate format during the transformation process. Assuming the QEP in Figure 4.2, a very *fine-grained* mapping might group tasks working on one operator, e. g., the probe operator P_1, into one TC. In contrast, a very *coarse-grained* mapping might group tasks processing an entire pipeline, e. g., pipeline $P1$ containing four operators, into one TC. Between these two extremes, there are several possible mappings for partial operator sequences.

In a third step, we use TCs to instantiate as many tasks as necessary to process all input tuples. Each task inherits an operator sequence, a buffer size, and all processing strategies from its task configuration. A task executes its operator sequence for each tuple in its buffer. Additionally, task execution is specified by its inherited processing strategies. Within a task, we encapsulate a particular piece of work of a QEP as an operator sequence and a subset of data as a chunk of tuples in a buffer.

The number of tasks per TC is determined by the ratio of input tuples and buffer size $\frac{input\_tuples}{buffer\_size}$. As the result of this transformation process, we obtain a set of tasks that is modeled in QTM. With QTM, we extend the notion of tasks proposed in previous work [Bea96, MOW97, LT92, Pea13] by a generalized work and data specification and a declaration of processing strategies for task execution during run-time.

4.2.2 Dynamic Load Balancing in QTM

With QTM, we model a QEP as a set of tasks. However, the actual scheduling of these tasks depends on the run-time implementation. A run-time implementation consists of two processing steps. First, it has to establish a particular order between tasks that satisfies the constraints introduced by a QEP. Second, it has to manage task execution following a scheduling strategy.

In this section, we introduce *QTM-DLB* as a run-time implementation of QTM. QTM-DLB implements a dynamic load balancing (DLB) approach with one global task queue. We decided to implement QTM using a DLB approach because it already based on the notion of tasks. Compared to other DLB approaches [Bea96, MOW97, LT92, Pea13], QTM-DLB executes generalized tasks specified in QTM. To establish a particular order between tasks, we define a *placement strategy* (PS). In QTM-DLB, we apply a placement strategy as the last step during compile-time to place tasks into the *global task queue*. Note, other possible run-time implementations might use the Volcano execution model or a run-time scheduler as a scheduling strategy. In this case, one task in QTM might represent a *next call* in the Volcano execution model or an *operator call* in a run-time scheduler.

The execution of tasks in a general task model is implemented by a mapping of tasks to UEs. In QTM-DLB, we choose to map tasks to threads because threads of the same process share an environment and allow for fast lightweight context switches. During run-time, the global task queue is processed sequentially from its beginning to its end by dequeuing one or multiple tasks by each UE. We assume, each UE is able to process each task and that all tasks are independent. Figure 4.3 illustrates query execution with QTM-DLB. At first, an UE dequeues a task from the head of the global task queue. In a next step, a task dequeues as many tuples as specified by its buffer size from an input queue, applies its operator sequence to each tuple,

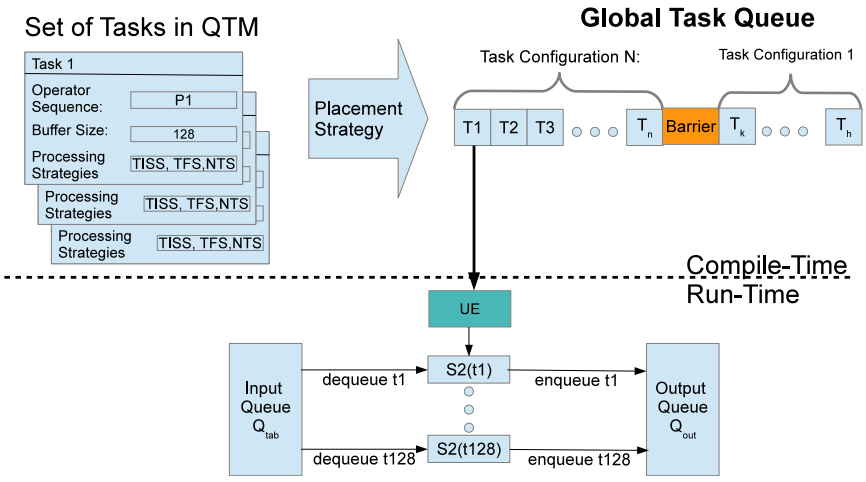


Figure 4.3: Query execution with QTM-DLB.

and enqueues qualifying tuples into an output queue. The distribution of tasks among UEs can be applied either statically or dynamically in a general task model. In QTM-DLB, UEs acquire tasks dynamically on their own if computing capacities are available.

The execution of tasks in a general task model requires that UEs are scheduled to PEs for execution. In QTM-DLB, this mapping differs for different DBMS. A DBMS running on a single machine may refer to one processor as one PE. Therefore, QTM-DLB would schedule UEs to processors. In contrast, a distributed DBMS may refer to one physical machine as one PE. Thus, QTM-DLB would schedule UEs to different machines. In this chapter, we focus on query execution on a single multi-core machine.

QTM and QTM-DLB are general enough such that all database scheduling strategies and chunk sizes shown in Figure 4.1 can be expressed in QTM and executed in QTM-DLB. We express different query execution strategies and chunk sizes with different task configurations, processing strategies, and placement strategies. Since QTM-DLB is based on a dynamic load balancing approach, it omits a run-time scheduler. Instead, QTM-DLB lays out a schedule during compile-time that is flexible enough to adapt itself to different run-time conditions. The actual schedule is determined by the dynamic run-time behavior of processors that acquire new work (tasks) on their own if computing capacities are available. In contrast, the Volcano execution model also omits a run-time scheduler but its scheduling is static and implicitly determined by its execution model.

For the rest of this chapter, we refer to QTM as our model that specifies query execution and QTM-DLB as a dynamic load balancing approach implementing QTM for query execution.

4.2.3 QTM Specification

In the following sections, we define QTM formally. We define task configurations in Section 4.2.3.1, processing strategies in Section 4.2.3.2, and queues in Section 4.2.3.3.

4.2.3.1 Task Configuration

In QTM, we define a task configuration (TC) that groups operators and tuples of a QEP. A task configuration TC_m is instantiated into n instances $\langle task_0^m \dots task_{n-1}^m \rangle$. For the rest of this chapter, we refer to instance i of a task configuration TC_m as $task_i^m$. Each TC specifies a buffer B of size b in tuples and an operator sequence O_l with operators $\langle o_0^l \dots o_{n-1}^l \rangle$ for its tasks. The operators in O_l satisfy a particular order. Each tuple t_i has to be processed by each operator $\langle o_0^l \dots o_{n-1}^l \rangle$ following the order of O_l. If tuple t_i has been deleted by operator o_i, then t_i will not be processed by the remaining operator sequence $\langle o_{i+1} \dots o_{n-1} \rangle$. Additionally, we define three processing strategies NTS, $TISS$, and TFS for a TC that specify run-time execution for its tasks (see Section 4.2.3.2). The number of instances per TC is defined by $\lceil \frac{number\ of\ input\ tuples}{buffer\ size} \rceil$. Each task is self-contained and includes all information necessary to execute the operator sequence for each tuple in its buffer.

4.2.3.2 Processing Strategies

In QTM, we define three processing strategies which specify run-time execution of tasks. All tasks of the same TC share the same *new tuple* strategy (NTS), *task internal scheduling* strategy ($TISS$), and a *tuple fetch* strategy (TFS). In the following, we present three QEP properties that require the definition of these processing strategies.

First, relational operators might create multiple output tuples from one input tuple. Thus, we define a *new tuple* strategy (NTS) for each TC. Following Manegold et al. [MOW97], we employ two strategies for handling new tuples. With NTS_{keep}, we refer to a strategy that keeps newly created tuples of operator o_i inside a task by adding them to its buffer. Thus, new tuples are processed by the following operator sequence $\langle o_{i+1} \dots o_{n-1} \rangle$. With NTS_{enq}, we refer to a strategy that creates new tasks for newly created tuples. Therefore, new tasks are inserted into the global task queue after the last task of the current TC. After that, the original task processes the remaining operator sequence. With NTS_{keep}, new tuples are kept on the same PE but the amount of work per task increases; thus, introducing an imbalanced task workload. On the other hand, with NTS_{enq}, the amount of work per task remains almost constant. However, newly created tasks are probably executed by another processor; thus, reducing data locality.

Second, if an operators sequence consists of more than one operator, different execution orders of tuples/operators are possible inside a task. Thus, we define a *task internal scheduling* strategy (TISS) for each TC. With $TISS_{op}$, task internal scheduling follows an *operator-at-a-time* approach such that all tuples $\langle t_0 \dots t_{n-1} \rangle$ are processed by operator o_i before the next operator o_{i+1} is applied. Using $TISS_{op}$, a TC processing a pipeline of c operators instantiates $c * \lceil \frac{number\ of\ input\ tuples}{buffer\ size} \rceil$ tasks with $c-1$ materializations between operators. With $TISS_{buf}$, task internal scheduling follows a *buffer-at-a-time* approach such that each tasks processes a chunk of tuples $\langle t_0 \dots t_{B-1} \rangle$ by each operator $\langle o_0 \dots o_{n-1} \rangle$. Using $TISS_{buf}$, a TC processing a pipeline of c operators instantiates $\lceil \frac{number\ of\ input\ tuples}{buffer\ size} \rceil$ tasks, each processing the entire pipeline. We do not model partial operator sequences inside tasks. If required, we would create different TC for each partial operator sequence.

Third, tuples inside a buffer can be accessed using different access strategies. Therefore, we define a *tuple fetch* strategy (TFS) for each TC. With TFS_{seq}, we refer to a strategy that fetches tuples sequentially inside each task. Consequently, operator o_i accesses tuples in sequential order $\langle t_0 \dots t_{B-1} \rangle$. With TFS_{zig}, we refer to a strategy that fetches tuples using a zig-zag access pattern. Thus, operator o_i accesses tuples in forward direction $\langle t_0 \dots t_{B-1} \rangle$ but operator o_{i+1} accesses tuples in backward direction $\langle t_{B-1} \dots t_0 \rangle$. Thus, TFS_{zig} might increase data locality for large data sets.

4.2.3.3 Queues

In QTM-DLB, we define a global task queue Q_{task} as a list of n tasks $\langle task_0 \dots task_{n-1} \rangle$ in a particular order. We refer to Q_{head} as the first element in Q_{task} that will be dequeued next. We refer to Q_{tail} as the last element in Q_{task}; thus, a new task will be enqueued at position Q_{tail+1}. During run-time, tasks are processed sequentially from Q_{head} to Q_{tail} following a first-in first-out approach.

We define three operations on Q_{task}. First, enq_{batch} inserts a batch of tasks $\langle task_0 \dots task_{n-1} \rangle$ starting at Q_{tail+1} following a placement strategy PS. This enqueue operation is used during compile-time. Second, $enq_{(task_i,pos)}$ inserts a single $task_i$ at position pos into Q_{task}. For example, NTS_{enq} requires this enqueue operation to insert newly created tasks into Q_{task} during run-time. Third, $dequeue_{num}$ dequeues the first num tasks starting from Q_{head}.

In QTM-DLB, we must satisfy the constraints introduced by a QEP. Thus, a synchronization point is required if TC_{m+1} depends on TC_m, i. e., all tasks of TC_m have to be processed before the first task of TC_{m+1} starts processing. Therefore, we define a barrier bar for Q_{task}. A barrier guarantees, that all tasks $\langle task_0^m \dots task_{n-1}^m \rangle$ of TC_m are processed before a $task_i^{m+1}$ of TC_{m+1} starts its execution.

Finally, we define three different data queues. Each input relation is modeled as a *table queue* Q_{tab}. Each table queue consists of n tuples $\langle t_0 \dots t_{n-1} \rangle$. Tuples in Q_{tab} are dequeued buffer-wise depending on the buffer size of the accessing task. Q_{int} defines an *intermediate* data queue for materialization. Note, each blocking operator and each barrier requires an implicit materialization of its result. With Q_{out}, we refer to a global output queue that stores the query result.

4.2.4 Parallelism in QTM

With QTM, we are able to express three forms of parallelism [GI96]. First, *partitioned parallelism* might be exploited by partitioning the input of an operator such that all partitions can be processed in parallel (intra-operator parallelism). In QTM, we model one TC for each partitionable operator and instantiate one task for each partition. Second, *pipelined parallelism* can be exploited by processing the entire pipeline without interruption or materialization. Note, operators in a pipeline are non-blocking and do not interfere with each other and thus enable *inter-operator parallelism*. In QTM, we model one TC containing the entire pipeline as an operator sequence. Additionally, we apply $TISS_{buf}$ for task internal scheduling. Third, *independent parallelism* or *inter-operator parallelism* can be exploited by executing independent pipelines in parallel. In QTM-DLB, we support independent parallelism by placing tasks from independent pipelines interleaved into the global task queue.

We optimize parallel query execution in QTM-DLB in four different ways. First, we improve *temporal locality* by grouping tuples into buffers and pipelines into operator sequences for tasks. Thus, we increase the probability for tuples to reside in cache for their entire processing. Furthermore, by processing tuples in chunks, we amortize the overhead per operator call through many tuples [Pea01, Bea99]. Second, we improve *spatial locality* by accessing tuples sequentially inside a buffer. The sequential access pattern leads to an increased cache line utilization and efficient prefetching. Third, we achieve a high *degree of parallelism* by specifying independent tasks that allow for asynchronous processing. Thus, independent tasks mitigate dependencies and reduce synchronization overhead. Fourth, we achieve high *resource utilization* by a loosely coupling of processing units and tasks.

4.3 Query Execution Schedules

In this section, we model common database schedules with QTM. Using a simple QEP, we demonstrate how common DBMS would implement different schedules. We show, that schedules mainly differ by their buffer size (chunk size) and their applied task internal scheduling strategy (TISS). For the following considerations, we assume one TC processing a pipeline of n

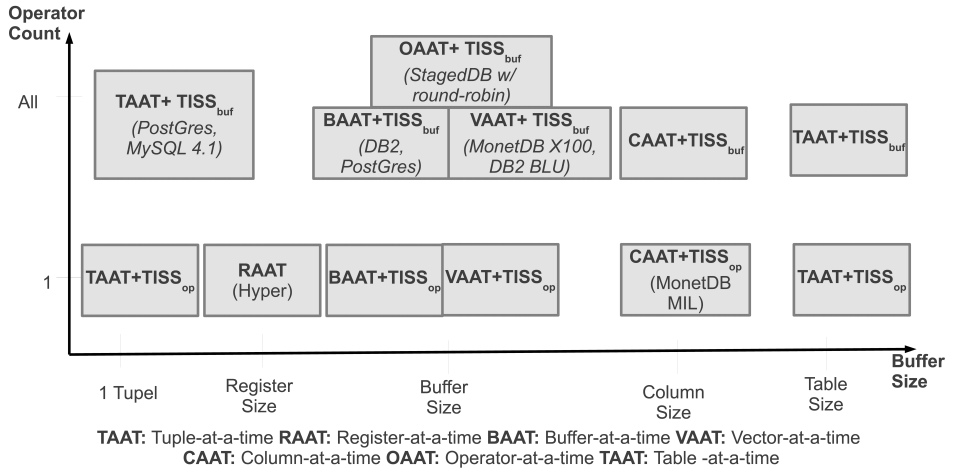


Figure 4.4: Query Execution Schedules in QTM.

operators. We omit NTS and TFS because they can be applied to any schedule. Figure 4.4 presents the schedules which are examined in this section. The buffer size refers to the number of tuples each task processes in a particular schedule. The operator count specifies the number of operators in the operator sequence of each task. For example, a task following a *tuple-at-a-time* + $TISS_{buf}$ schedule would process the entire pipeline with one tuple. In contrast, a task following a *tuple-at-a-time* + $TISS_{op}$ schedule would process only one operator with one tuple.

A *tuple-at-a-time* (TAAT) schedule performs one operator call for each tuple. The *Volcano execution model* is one common example implementing this schedule [Gra90]. It is used in MySQL, PostGres, and System R. We model a tuple-at-a-time schedule by defining a buffer size of one. To support row and column-oriented storage layouts, we utilize a `fetch` function for each operator call to fetch the next tuple t_{i+1}. The actual implementation of the `fetch` function differs depending on the storage layout. For a row-oriented storage layout (NSM), one memory access returns the entire tuple. For a column-oriented storage layout (DSM), the fetch function collects all required attributes from v columns, thus resulting in v memory accesses. Considering performance, one operator call per tuple results in a large overhead due to many operator calls. We identify two possible operator sequences. With $TISS_{buf}$, a task processes one tuple t_i by all operators $\langle o_0 \dots o_{n-1} \rangle$. This schedule is used by a Volcano execution of a pipeline of operators. With $TISS_{op}$, each task executes one operator o_i for one tuple t_i. However, o_i has to be processed entirely for all tuples $\langle t_0 \dots t_{n-1} \rangle$ before o_{o+1} starts processing. Thus, with $TISS_{op}$, operators are processed in a step-wise manner which requires materialization of intermediate results. This schedule is used by a Volcano execution of a blocking operator. In contrast, with $TISS_{buf}$, tuples are only materialized while percolating the pipeline.

A *register-at-a-time* (RAAT) schedule was introduced by Neumann and implemented in Hyper [Neu11]. This schedule combines operators inside the same pipeline into one operator. The *combined* operator processes as many tuples as fit into one CPU register. Therefore, the buffer size depends on the size of a CPU register and the size of a tuple. The combined operator reduces the number of operator calls to one call per pipeline per buffer. Therefore, the overhead per operator call is amortized over all tuples in the buffer and over all operators in the pipeline. Since the pipeline is compressed to only one operator per pipeline, only one possible execution order exists. Thus, we omit $TISS$ in Figure 4.4. Although Neumann [Neu11] evaluates this approach for DSM, it would also be applicable to NSM.

A *buffer-at-a-time* (BAAT) schedule performs one operator call for each buffer. In general, the buffer can be of any size. However, previous work shows that a buffer size that matches a hardware parameter exhibits the best performance [Zea08, Pea01, CRG07, ZR04]. Common examples are the size of the L1, L2 or L3 cache. DB2 5.2 [Pea01] as well as PostgreSQL 7.3.4 [ZR04] implement this buffer-at-a-time schedule. In addition to these static buffer sizes, Cieslewicz et al. [Cea09] introduce a buffer that changes its size dynamically based on cache miss sampling. Additionally, we have to take the storage layout into account. A buffer storing tuples of a NSM storage layout consists of the entire tuple with all attributes. In contrast, a buffer storing tuples of a DSM storage layout usually consists of attribute values for a single column. In our QTM model, we model the buffer-at-a-time schedule only for the NSM storage layout and leave the DSM storage layout for the vector-at-a-time schedule. The operator sequences are similar to the tuple-at-a-time schedule. However, instead of processing one tuple, a task following $TISS_{buf}$ processes all tuples in its buffer B at operator o_i before processing the same buffer at the next operator o_{i+1}. With $TISS_{op}$, a task processes one buffer B with one operator o_i. Again, o_i has to be processed entirely for all buffers $\langle B_0 \dots B_{n-1} \rangle$ before o_{o+1} starts processing and thus materialization is required. Considering performance, the overhead per operator call for $TISS_{buf}$ and $TISS_{op}$ is amortized over all tuples in the buffer. Thus, the advantages of the block-oriented processing [Rea13, Pea01] are exploited. Additionally, tasks following $TISS_{buf}$ amortize their overheads over all operators in the pipeline.

A *vector-at-a-time* (VAAT) schedule performs one operator call for each vector of each column. This schedule is implemented by MonetDB/X100, C-Store, and DB2 with BLU. MonetDB/X100 [Bea05] and DB2 with BLU [Rea13] adjust their buffer size such that all data are cache resident. In contrast, C-Store [Sea05] processes blocks of 64 KB. The vector-at-a-time schedule is essentially a buffer-at-a-time schedule, but introduces one buffer per column. In contrast, a buffer-at-a-time schedule introduces one buffer for the entire relation or between operators. Additionally, a buffer-at-a-time schedule determines the buffer size in relation to the size of an entire

tuple. In contrast, a vector-at-a-time schedule has to determine a separate buffer size for each column in relation to the size of the attribute values. The number of buffers increases with each accessed column. One major advantage of a vector-at-a-time schedule is its opportunity for *vectorized processing*. Vectorized processing enables SIMD processing that showed an improved performance [Zea08, Bea05]. Another important advantage of a vector-at-a-time schedule is its increased buffer utilization if only a small fraction of all attributes are accessed. In contrast, a buffer-at-a-time schedule on a NSM storage layout would load unused data into its buffer if only a small fraction of all attributes are accessed. The operator sequences are similar to the buffer-at-a-time schedule but extend one call per buffer to one call per buffer per column. The processing of $TISS_{buf}$ and $TISS_{op}$ inside the operator sequences remains unchanged, but an additional call for each column is added. Note, the processing of different columns per operator introduces an opportunity for scheduling columns in different orders.

A *column-at-a-time* (CAAT) schedule performs one operator call for each column. MonetDB/MIL [Bea99] implements this type of schedules. It requires a DSM storage layout and corresponds to a vector-at-a-time approach with the entire columns as one vector. However, when executing a column-at-a-time schedule using multiple PEs, the entire column may be partitioned into chunks, i. e., this schedule transforms into a vector-at-a-time schedule. Processing an entire column introduces additional costs for materialization of intermediate results, thus increasing the memory consumption [Bea99]. The buffer size corresponds to the number of tuples in a column. With $TISS_{buf}$, a task processes one column col_i entirely with operator o_i before processing the same column with the next operator o_{i+1}. With $TISS_{op}$, a task processes col_i with only one operator o_i and all columns $\langle col_0 \dots col_{n-1} \rangle$ have to be processed by o_i before o_{o+1} starts processing.

A *table-at-a-time* (TAAT) schedule performs one operator call for the entire table and can be found in OLTP databases that apply a *data manipulation operation* to an entire table. To support a row-oriented and column-oriented storage layout, we utilize the `fetch` function introduced for a tuple-at-a-time schedule. When executing a table-at-a-time schedule using multiple PEs, the entire table can be partitioned into chunks, i. e., this schedule transforms into a buffer-at-a-time schedule. The processing with $TISS$ are similar to a buffer-at-a-time schedule with one buffer for the entire table.

An *operator-at-a-time* (OAAT) schedule represents a special schedule that follows the *StagedDB* approach. This schedule is implemented in STEPS and QPipe [HA05]. An operator sequence is divided into stages that represent operators. The buffer size corresponds to the size of an input queue at each stage. The stages exchange tuples via messages from one input queue to another. While not stated, we assume STEPS and QPipe work on a NSM storage layout because the prototypes are based on Shore and BerkeleyDB

which use a NSM storage layout [HA05]. The actual operator sequence depends on the applied scheduling algorithm. A simple round-robin scheduling will call each operator for a fix time slice before calling the next in a circular manner. However, due to back pressure or other scheduling decisions, an arbitrary operator sequence is possible. Based on the scheduling algorithm, each stage processes tuples in its input queue as long as its time slice is valid or until its input queue becomes empty.

In Table 4.1, we summarize our classification of common approaches for query execution using our QTM model. We assume a pipeline containing m operators $o_0 \dots o_{m-1}$, n tuples $t_0 \dots t_{n-1}$, and k columns $col_0 \dots col_{k-1}$. Furthermore, we assume that v attributes of each tuple t_i are accessed. We describe each approach and show at least one DBMS implementing this approach in Table 4.1. Furthermore, we define the buffer size and operator sequence. Finally, we show its applicability for NSM or DSM data layout. Note, operator sequences and buffer sizes are independent of the number of processing units; thus, excluding parallelism. In general, tasks are self-contained work packages which contain all information necessary to execute its processing independently. Their order of execution depends on the assignment of tasks to PEs.

Approach	**Description**	**DBMS**	**Buffer Size** B	**Data Layout**	**Operator Sequence**
Tuple-at-a-time	One operator call for each tuple	MySQL4.1, Volcano-style [Gra94, Gra90]	$B = 1$, one tuple	NSM DSM	$TISS_{buf} : \{o_0(t_0) \dots o_{m-1}(t_0)\} \dots \{o_0(t_{n-1}) \dots o_{m-1}(t_{n-1})\}$
					$TISS_{op} : \{o_0(t_0) \dots o_0(t_{n-1})\} \dots \{o_{m-1}(t_0) \dots o_{m-1}(t_{n-1})\}$
Register-at-a-time	One combined operator call for each register buffer.	Hyper [Neu11]	$\frac{sizeOf(Register)}{sizeOf(tuple)}$	NSM DSM	$TISS_{buf} : o_{combined}(t_0 \dots t_{B-1}) \dots o_{combined}(t_{n-B} \dots t_{n-1})$
Buffer-at-a-time	One operator call for each buffer. Buffer size varies.	PostgreSQL 7.3.4 [ZR04], DB2 5.2 [Pea01],[Cea09, CRG07]	$\frac{sizeOf(HWPar)}{sizeOf(tuple)}$	NSM	$TISS_{buf} : \{o_0(t_0 \dots t_{B-1}) \dots o_{m-1}(t_0 \dots t_{B-1})\} \dots \{o_0(t_{n-B} \dots t_{n-1}) \dots o_{m-1}(t_{n-B} \dots t_{n-1})\}$
					$TISS_{op} : \{o_0(t_0 \dots t_{B-1}) \dots o_0(t_{n-B} \dots t_{n-1})\} \dots \{o_{m-1}(t_0 \dots t_{B-1}) \dots o_{m-1}(t_{n-B} \dots t_{n-1})\}$
Vector-at-a-time	One operator call for each vector in each column. Vector size varies.	MonetDB/ X100 [Bea05] Vector size = cache size C-Store [Sea05] 64K block, DB2 with BLU [Rea13]	$\frac{sizeOf(HWPar)}{sizeOf(attribute)}$	DSM	$TISS_{buf} : [col_0\{o_0(t_0 \dots t_{B-1}) \dots o_{m-1}(t_0 \dots t_{B-1})\} \dots \{o_0(t_{n-B} \dots t_{n-1}) \dots o_{m-1}(t_{n-B} \dots t_{n-1})\}] \dots [col_{k-1}\{o_0(t_0 \dots t_{B-1}) \dots o_{m-1}(t_0 \dots t_{B-1})\} \dots \{o_0(t_{n-B} \dots t_{n-1}) \dots o_{m-1}(t_{n-B} \dots t_{n-1})\}]$
					$TISS_{op} : [col_0\{o_0(t_0 \dots t_{B-1}) \dots o_0(t_{n-B} \dots t_{n-1})\} \dots \{o_{m-1}(t_0 \dots t_{B-1}) \dots o_{m-1}(t_{n-B} \dots t_{n-1})\}] \dots [col_{k-1}\{o_0(t_0 \dots t_{B-1}) \dots o_0(t_{n-B} \dots t_{n-1})\} \dots \{o_{m-1}(t_0 \dots t_{B-1}) \dots o_{m-1}(t_{n-B} \dots t_{n-1})\}]$
Column-at-a-time	k operator calls for k columns	MonetDB/ MIL [Bea99]	$sizeOf$ $(column)$	DSM	$TISS_{buf} : \{o_0(col_0) \dots o_{m-1}(col_0)\} \dots \{o_0(col_{k-1}) \dots o_{m-1}(col_{k-1})\}$
					$TISS_{op} : \{o_0(col_0) \dots o_0(col_{k-1})\} \dots \{o_{m-1}(col_0) \dots o_{m-1}(col_{k-1})\}$
Operator-at-a-time	One stage per operator. Each stage one stage buffer SB.	StagedDB[HA03], QPipe [HSA05], STEPS [HA04]	$sizeOf(SB)$	NSM (DSM)	Depending on the scheduling algorithm. In case of round-robin using $TISS_{buf}$:$stage_0(t_0 \dots t_{SB-1}) \dots stage_{n-1}(t_0 \dots t_{SB-1})$ In case of sequential using $TISS_{op}$: $\{o_0(t_0) \dots o_0(t_{n-1})\} \dots \{o_{m-1}(t_0) \dots o_{m-1}(t_{n-1})\}$

Table 4.1: State-of-the-art classification.

4.4 Evaluation

In this section, we evaluate different schedules that are modeled in QTM and executed in QTM-DLB. At first, we describe our experimental setup in Section 4.4.1. After that, we introduce our test schedules in Section 4.4.2. Then, we compare these schedules with respect to run-time in Section 4.4.3 and resource utilization in Section 4.4.4. Finally, we examine their scalability Section 4.4.5.

4.4.1 Experimental Setup

We present our experimental setup in the following. We describe our prototype in Section 4.4.1.1, the workload in Section 4.4.1.2, and the used hardware and software in Section 4.4.1.3.

4.4.1.1 Prototype

We implement QTM-DLB as a prototype in C++. QTM-DLB executes queries modeled in QTM (see Section 4.2.2). In a preparation step, we create a set of tasks and place them into a global task queue. The order of tasks and their configuration represent a schedule. We exclude the preparation step for our measurements and measure solely query execution during run-time. Query execution in QTM-DLB proceeds as follows. At first, each processor dequeues a tasks from the global task queue. After that, each task dequeues all tuples for its processing from an input or intermediate queue into its buffer and applies its operator sequence to each tuple. The processing strategies of each task specify the execution order of operators, the access pattern for tuples in its buffer, and the processing of newly created tuples. Finally, each task enqueues qualifying tuples into a global output or intermediate queue. This sequence is repeated until all tasks in the global task queue are processed.

Although tasks run asynchronously and mainly process *task-local* data in their buffers, they have to synchronize on shared data structures. In our prototype, we have to synchronize 1) dequeuing of tasks from the global task queue, 2) dequeuing of tuples from an input or intermediate queue, and 3) enqueuing of result tuples into a intermediate or global output queue. In QTM-DLB, we synchronize these three queue operations with atomic counters as proposed by Cieslewicz et al. [CRG07]. Thus, each enqueuing or dequeuing operation of n tasks or tuples increments an atomic counter by n. After that, a task can exclusively access tasks or tuples from n_{old} to n_{new-1}. Note, these three synchronization operations represent the overhead introduced for each task in QTM-DLB.

Within each task, bookkeeping of qualified tuples among different operators is maintained by a bitmask. Bit i of a bitmask represents the qualification of tuple i in buffer B. Each operator applies it processing only if

tuple i was qualified by previous operators. Then, each operator updates the bitmask using an `AND` operation for its qualified tuples. Finally, the last operator in a pipeline places all qualified tuples into the global output or intermediate queue. We exclude tuple modification inside a pipeline, e. g., a concatenation of two attribute values in our QTM model.

For this evaluation, we implement a selection operator and a hash join using a shared hash table. Each tuple in an input relation consists of an 8 byte key and an 8 byte payload. The hash join is implemented as a non-partitioning hash join following Blanas et al. [Bea11a] with the improvement of an contiguous array for buckets proposed by Balkesen et al. [Bea13]. Each hash table consists of small buckets with 32 entries per bucket. Each bucket entry consists of an 8 byte key and an 8 byte pointer. We implement the same the hash function as used in PostgreSQL [Pos17].

4.4.1.2 Workload

In our evaluation, we model different schedules for the example QEP shown in Figure 4.5. The QEP consists three input relations, one selection operator, and two hash based equi-joins. The dataset is synthetically generated and consists of three relations containing 30M tuples in ascending order. We introduce skew by incrementing tuples with different values as shown in the table in Figure 4.5. As a result, each join has a selectivity of 50%. Furthermore, the selection at the beginning of the pipeline filters 5M tuples.

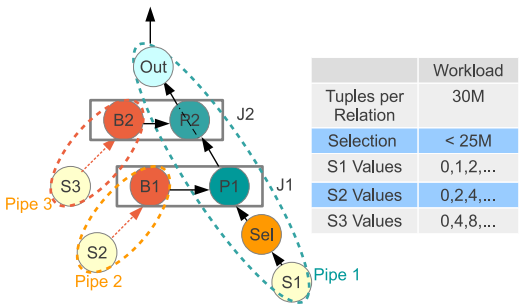


Figure 4.5: TestCase: Multi-Level Join.

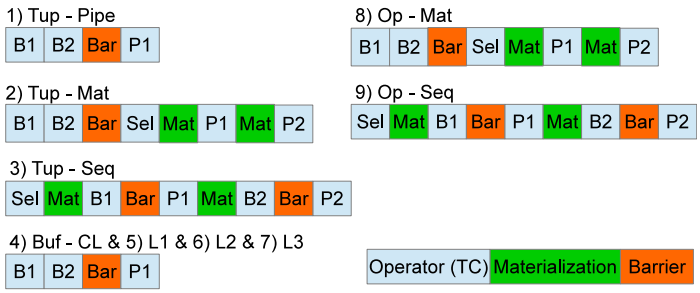
4.4.1.3 Hardware and Software

We evaluate our prototype on an Intel Xeon E7-4870 CPU. The CPU contains 10 physical cores, each supporting *hyper-threading*. The cache hierarchy of each core is composed of one separate 32 KB L1 cache for instructions and data, each 8-way set associative. Additionally, each core owns a unified 256 KB L2 cache for data and instructions, each 8-way set associative. The L1 and L2 cache are exclusive to each core. Finally, all cores share a 32 MB

24-way set associative L3 cache. We ran our experiments on an *openSUSE 13.1* using a 3.14.4 kernel. Our prototype was compiled with GCC 4.8.1 using O3 compiler optimizations. We measure performance counters using the PAPI framework [PAP17].

4.4.2 Test Schedules

For our evaluation, we implemented nine different schedules for the QEP shown in Figure 4.5. In Figure 4.6a, we illustrate these schedules as a sequence of operators. We model one operator as one TC and instantiate tasks as shown in Figure 4.6b. In general, the buffer size determines the number of tasks per operator and is either fixed (Schedule 1-3), matches a cache size (Schedule 4-7), or is determined in relation to the current DOP (Schedule 8-9). In contrast, the scheduling strategy determines the number and order of operators as well as the number of materializations and barriers.



(a) Test Schedules.

X-At-A-Time Approaches	Schduling Type	Task-internal Scheduling (TISS)	Buffer Size in Tuples	Tasks per Op	Total Tasks
1) Tup - Pipe	T-AAT	TISS(buf)	1	30M	90 M
2) Tup - Mat	T-AAT	TISS(op)	1	30M	150 M
3) Tup - Seq	T-AAT	TISS(op)	1	30 M	150 M
4) Buf - CL	B-AAT	TISS(buf)	4	7.5 M	22,5 M
5) Buf - L1	B-AAT	TISS(buf)	2048	14.649	43.947
6) Buf - L2	B-AAT	TISS(buf)	16384	1.832	5.496
7) Buf - L3	B-AAT	TISS(buf)	491.520	62	186
8) Op - Mat	O-AAT	TISS(op)	7.5 M	4	20
9) Op - Seq	O-AAT	TISS(op)	7.5 M	4	20

(b) Test Configurations.(DOP=4)

Figure 4.6: Test Cases.

We model Schedules 1-3 in QTM with different *task-internal scheduling strategies* (*TISS*) as *tuple-at-a-time* schedules (T-AAT). Schedules 1-3 represent three possible schedules for the Volcano execution model using a buffer size of one. Since each operator instantiates one task per tuple, 30 million tasks per operator are created. However, the total number of operators differ

between Schedules 1-3 due to different execution orders. Tasks in Schedules 1 and 2 build hash tables $B1$ and $B2$ for relations $S2$ and $S3$ until the barrier is reached. The barrier satisfies the constraint that the first probe operator has to wait until all hash tables are built entirely. We model Schedule 1 in QTM with $TISS_{buf}$. Thus, each task processes the entire pipeline for one tuple. In contrast, we model Schedules 2 and 3 in QTM with $TISS_{op}$. Thus, all tasks cooperatively finish the processing of one operator and materialize their results before processing the next operator. Note, materialization eliminates pipeline parallelism and increases the number of tasks due to an increased number of TCs. As shown in Figure 4.6a, we combine a table scan (S2 or S3) and a hash build (B1 or B2) into one operator $B1$ or $B2$. Thus, $Pipe2$ and $Pipe3$ are reduced to one operator. A pipeline containing only one operator allows only one possible execution order. Thus, $TISS$ did not affect execution order of $B1$ or $B2$.

In contrast to Schedule 1 and 2, we model Schedule 3 in QTM with $TISS_{op}$ as a schedule executing a sequential join order; thus, joins are not interleaved. The execution order changes to 1) applying the selection to each tuple in $S1$, 2) building the hash table for $S2$ and probing the intermediate result of the selection ($S1$) in $B1$, and 3) building the hash table for $S3$ and probing the intermediate result of the previous probe ($P1$) in $B2$. As shown in Figure 4.6a, the sequential join order changes the execution order and increases the number of barriers. However, the total number of tasks remains equal to Schedule 2.

We model Schedules 4-7 in QTM with $TISS_{buf}$ as *buffer-at-a-time* schedules (B-AAT). The buffer sizes match different cache sizes. Schedules 4-6 determine their buffer size such that all tuples fit into a cache line (Schedule 4), L1 cache (Schedule 5), or L2 cache (Schedule 6). Schedule 7 divides the L3 cache between the number of executing threads (DOP). Thus, the buffer size is determined by $\lceil \frac{size\ of\ L3}{DOP} \rceil$. Similar to Schedule 1, Schedule 4-7 exploit pipeline parallelism but execute $Pipe1$ for a chunk of tuples. As shown in Figure 4.6b, an increased buffer size reduces the number of tasks per operator. The total number of tasks ranges from 186 to 22,5M. Note, Schedules 4-7 are common in the Volcano execution model using block-oriented processing or in a cache-conscious run-time scheduler.

We model Schedules 8 and 9 in QTM with $TISS_{buf}$ as *operator-at-a-time* (O-AAT) schedules. The buffer size is determined by dividing the input tuples equally between available threads. As a result, the number of tasks is equal to the number of threads (DOP). Schedules 8 and 2 as well as 9 and 3 model the same execution order and number of operators. However, the buffer sizes differ significantly. Schedules 2 and 3 model the smallest possible buffer size of one tuple and Schedule 8 and 9 model the largest buffer with regard to DOP. These different buffer sizes impact the number of tasks significantly. Schedules 8 and 9 might be found in MonetDB [Bea99].

4.4.3 Run-time

In Figure 4.7, we show run-times in nanoseconds per input tuple for each test schedule presented in Section 4.4.2. We execute each schedule in QTM-DLB with a DOP of four. Figure 4.7 shows, that B-AAT Schedules 5-7 achieve the shortest run-times and T-AAT Schedules 1-3 the longest. Furthermore, O-AAT Schedules 8 and 9 are slightly slower that the B-AAT Schedules 5-7 but faster than B-AAT Schedule 4.

Schedule 1 implements the most efficient T-AAT schedule that reduces the run-time by a factor of almost two compared to Schedules 2 and 3. The main reasons are the reduced number of tasks and the exploitation of pipeline parallelism. Schedules 2 and 3 execute the same number of tasks, materialize the same number of intermediate results, but do not exploit pipeline parallelism. However, they model a different execution order that shows only marginal impact on run-time.

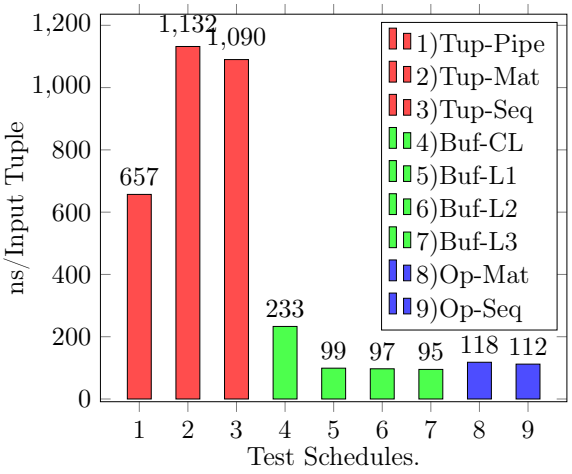


Figure 4.7: Run-times for Test Schedules. (DOP = 4)

B-AAT Schedules 4-7 model differently sized buffers which result in different numbers of tasks. Schedule 4 executes 22,5M tasks, each loading as many tuples as fit into a cache line (4 tuples). Although, Schedule 4 decreases the run-time by a factor of five compared to Schedules 2 and 3, the large number of tasks results in the longest run-time of all B-AAT schedules. Schedules 5-7 decrease the run-time by a factor of two compared to Schedule 4 and by a factor of ten compared to Schedules 2 and 3. However, run-times for Schedule 5-7 are very similar. Schedule 5 executes about 40K tasks and performs slightly worse than Schedule 6 executing roughly 5K tasks. Schedule 7 achieves the best overall run-time by executing only 186 tasks.

O-AAT Schedules 8 and 9 execute fewer tasks than all other schedules (only 20 tasks). As in Schedules 1-3, the impact of a different execution

order on run-time is only marginal. However, Schedules 8 and 9 with very large buffers improve run-time by a factor of ten compared to Schedules 2 and 3 executing the same schedule with very small buffers. In the next section, we present explanations for these different run-times by sampling query execution.

4.4.4 Time Distribution

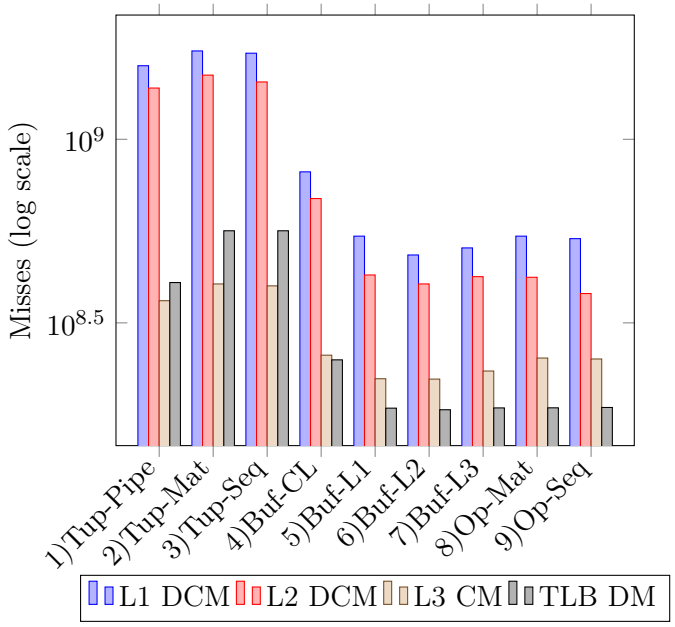
We analyzed the utilization of the cache hierarchy by our test schedules to explain run-times differences. Our findings revealed, that the cache hierarchy impacts the run-times to a high degree. Figure 4.8a and Figure 4.8b present our sampling results. Additionally, Figure 4.9 shows a breakdown of misses. In these figures, a performance counter samples either data cache misses (DCM), instruction cache misses (ICM), or misses in a unified instruction and data cache (CM). Additionally, we measure *Translation Lookaside Buffer* misses for data pages (TLB DM) and instruction pages (TLB IM) as well as the number of branch miss predictions (Branch MP). We present our general observations in Section 4.4.4.1. Furthermore, we describe data cache behavior in Section 4.4.4.2 and instruction cache behavior in Section 4.4.4.3. Finally, we summarize our results in Section 4.4.4.4.

4.4.4.1 Observations

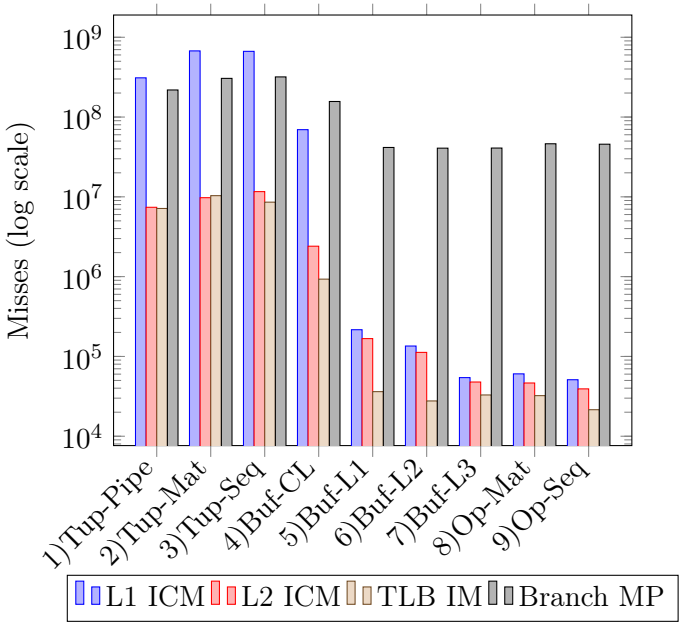
As a first observation, Schedules 2 and 3 as well as 8 and 9 reflect similar counter values in Figure 4.8a and Figure 4.8b. These similar numbers of cache and TLB misses explain similar run-times observed in Figure 4.7. However, the sequential join execution of Schedules 3 and 9 causes slightly less misses and thus improves run-time marginally.

As a second observation, Schedule 1 improves run-time compared to Schedules 2 and 3 which can be attributed to less data and instruction cache misses. The main reasons for that are threefold. First, Schedule 1 exploits pipeline parallelism which reduces the number of data cache misses (L1 DCM & L2 DCM). Second, Schedule 1 executes less tasks which reduces the number of instructions and instruction related cache misses (L1 ICM & L2 ICM). Third, following an improved data and instruction cache utilization, TLB misses for data and instructions (TLB DM & TLB IM) are reduced as well as the number of L3 cache misses (L3 CM) and branch mispredictions (Branch MP).

We further observe that data cache misses of Schedules 8 and 9 are similar to Schedule 5. The small number of four tasks per operator for Schedule 8 and 9 results in the fewest ICMs. However, eliminating pipeline parallelism and the requirement of materialization result in longer run-times compared to B-AAT schedules.



(a) Data Related Misses.



(b) Instruction Related Misses.

Figure 4.8: Cache Misses.

The fourth observation is contrary to the general assumption that a buffer that fits entirely into a private cache exhibit less data cache misses [Zea08, Pea01, CRG07, ZR04]. As shown in Figure 4.8a, this assumption does not

hold for Schedules 1 and 4 using very small buffer sizes. Besides the huge number of tasks, the reasons for that are twofold. First, Schedule 1 cannot exploit spatial locality because one cache line is shared among different tasks. Thus, each cache line is loaded multiple times. Second, a small buffer size prevents efficient prefetching. For example, if Schedule 4 is executed by n threads, each tasks loads every n-th cache line (omitting the dynamic runtime behavior). In contrast, a task in Schedule 5 accesses 512 cache lines sequentially. Thus, a hardware prefetcher may detect the access pattern of Schedule 5 but not the access pattern of Schedule 4.

Finally, the breakdown misses in different caches in Figure 4.9 reveals a different distribution among schedules. Note, some misses are hidden because of their marginal occurrence. The observations are five-fold. First, L2 DCM are more frequent than other misses. Second, TLB DM and L2 DCM are almost constant over all schedules. Third, L2 ICM and TLB IM are negligible. Fourth, L3 CM are more frequent and Branch MP are less frequent for larger buffer sizes (Schedules 4-9). Fifth, L1 ICM are more frequent for smaller buffer sizes (Schedules 1-4). Note, a time breakdown can be derived from Figure 4.9 by multiplying the number of cache misses with the actual miss penalty.

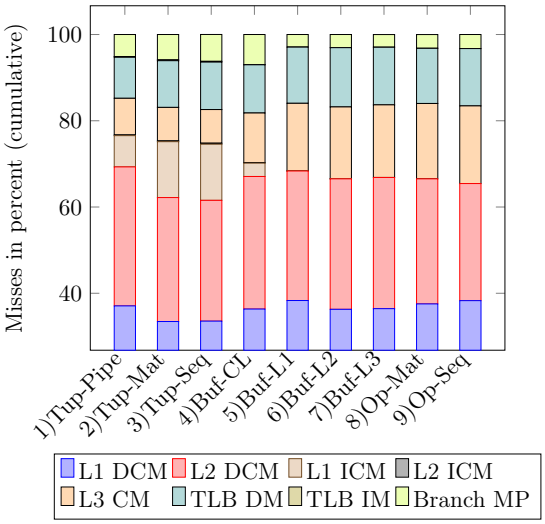


Figure 4.9: Breakdown of Misses.

4.4.4.2 Data Cache Misses

Our sampling results for Schedules 4-7 in Figure 4.8a show, that data caches misses in L1, L2, and L3 cache decrease with increasing buffer size until the buffer size exceeds the largest private cache (L2). Beyond that size, data cache misses increase. The main reason for that originates from a

different exploitation of pipeline parallelism in different schedules. Pipeline parallelism enables data locality for tuples percolating the pipeline. In an optimal case, the first operator in a pipeline loads all tuples into the cache. Then, each consecutive operator will induce no cache miss because its data is already loaded. The number of cache misses is reduced as long as the data set fits into the cache. Unfortunately, this optimal case requires that only the first operator in a pipeline loads the entire data set. However, this requirement is usually not satisfied because consecutive operators like a hash probe may also load data into the cache. Each additional data load increases the probability that already loaded tuples are evicted before reuse (so-called *cache-thrashing*). As shown in Figure 4.8a, cache trashing occurs as soon as the buffer size exceeds the private L2 cache. Starting from this buffer size, cache misses increase up to a point where each data access results in a cache miss and no data locality inside the pipeline can be exploited. The TLB data cache misses follow this trend.

4.4.4.3 Instruction Cache Misses

Our sampling results in Figure 4.8b show, that instruction cache misses are correlated with the number of tasks. Thus, schedules processing tasks with large buffers (Schedules 7-9) decrease the total number of tasks and amortize their task overhead among multiple tuples. Additionally, they increase the locality of instructions by processing multiple tuples in tight loops. The improved instruction locality results in less instruction cache misses.

Compared to data cache misses, instruction cache misses are more performance critical because they cannot be overlapped using *out-of-order* execution [HA05]. In the worst case, the processor pipeline stalls until the instructions are fetched. This is the main reason why Schedule 7 is faster than Schedule 6. Although Schedule 7 induces more data cache misses, the reduced number of performance critical instruction cache misses are more crucial for the overall run-time. TLB instruction misses and branch miss predictions follow the characteristics of the instruction cache. Finally, small tasks result in more branch mispredictions because the number of branch targets are increased which pollutes the *branch target buffer*.

4.4.4.4 Results

As a result of our evaluation using performance counters, we identify a trade-off between data and instruction cache performance. We show, that a schedule that is optimized for data cache locality does not necessarily outperform a schedule that is optimized for instruction cache locality. Overall, the cache performance can be adjusted by the buffer size which impacts data cache as well as instruction cache performance. We reveal, that a schedule that produces medium sized tasks (Schedules 4-6) by determining its buffer size

based on a cache size exploits data locality most efficiently. On the other hand, the high number of tasks introduce many instructions; thus, causing many instruction cache misses. In contrast, a schedule that produces large tasks (Schedules 7-9) cause less instruction cache misses due to a decreased number of instructions; thus, exploiting instruction locality efficiently. Finally, few large tasks cause more data cache misses if the buffer size exceeds the data cache size.

4.4.5 Scalability

In Figure 4.10, we examine the scalability of our test schedules. We exploit between one and 20 cores. Starting from a DOP of ten, hyper-threading is applied. In general, hyper-threading interleaves threads at a fine granularity and is beneficial if two threads execute different types of work. For example, if one thread handles an *I/O* request and the other executes computation [ZCRS05]. Although hyper-threading introduces two logical cores per physical core, both cores have to share many execution resources, including memory bus and caches [ZCRS05].

T-AAT Schedules 2 and 3 scale up to a DOP of 11, then stagnate between 12-14 cores before increasing run-time starting from 15 cores. However, the best reported speedups of nearly 1.4 with 13 cores for both schedules is marginal. Even worse, with 20 cores, Schedules 2 and 3 are nearly as fast as running the same schedule with only two cores. Schedule 1 scales with the same characteristics but exhibits a slightly larger speedup of two. The reasons for the poor scalability of the T-AAT schedules are threefold. First, Schedules 1-3 cannot produce enough independent work to overlap data cache misses with useful computation. Schedule 1 scales slightly better because it exhibits less cache misses and thus frees up memory bandwidth that is available for other cores. However, Schedules 1-3 are *memory-bound*. Second, threads execute tasks that perform similar work on different data. If executing two threads on the same core using hyper-threading, both threads require the same execution units and thus the benefit of hyper-threading cannot be exploited to its full extent. Furthermore, the available resources per thread, e. g., caches, are divided by two and threads may evict tuples mutually. Third, context switches between threads are less expensive with hyper-threading but still introduce some overhead.

B-AAT Schedules 4-7 scale much better and do not increase run-time if hyper-threading is applied. The main reason for that are the more efficient exploitation of the cache hierarchy. Therefore, more data accesses can be overlapped with computation and more data accesses can be satisfied by private caches. Schedule 6 achieves the highest speedup (8.3), followed by Schedule 7 (7.4), Schedule 5 (7.2), and Schedule 4 (3.6). The shortest run-time is achieved by Schedule 6 with a DOP of 20. However, Schedules 6 and 7 compete for the best run-time. Schedule 6 achieves shorter run-times for

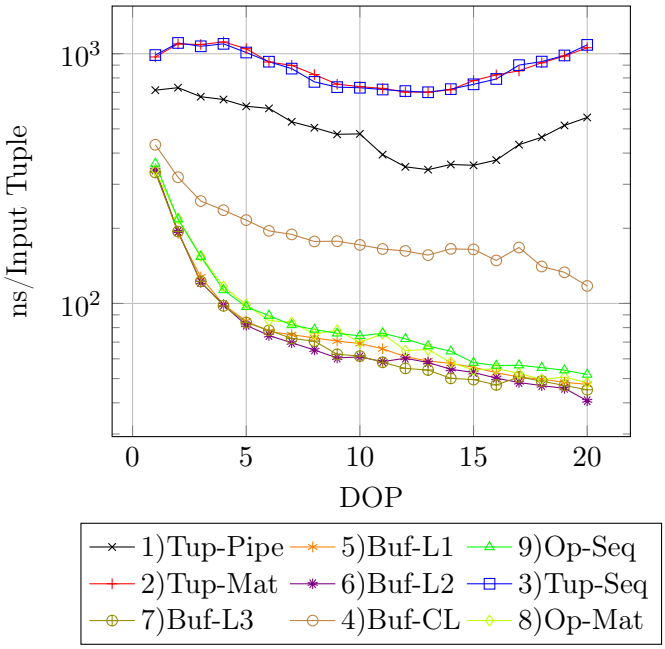


Figure 4.10: Scalability of Test Schedules.

a DOP less than ten and Schedule 7 for a DOP larger than ten. Overall, Schedule 6 achieves the best run-time in 11 of 20 samples. The reason for this competition is the trade-off between data and instruction cache misses as described in Section 4.4.4. Therefore, Schedule 6 produces less data but more instruction cache misses and Schedule 7 produces less instruction but more data cache misses.

Schedules 8 and 9 exhibit same characteristics as Schedules 5-7 but with slightly longer run-times and less speedup of 7.1 for Schedule 8 and 6.9 for Schedule 9. The difference can be attributed to the elimination of pipeline parallelism and the materialization of intermediate results.

As a result, we identify a sweet spot where the ratio of data locality and instruction locality produces the fastest schedules. The sweet spot lies between a schedule with a buffer size matching the largest private cache (Schedule 6) and a schedule with a slightly larger buffer size that reduces instruction cache misses (Schedule 7).

4.5 Summary

In this chapter, we classified common databases by their scheduling strategy and chunk size. Furthermore, we introduced a *Query Task Model (QTM)* that allows us to express and compare different approaches for parallel query execution. With QTM, we open a design space for database schedules. In our

evaluation, we examined how different schedules exploit resources of modern CPUs. We showed, that a schedule that is optimized for data cache locality does not necessarily outperform a schedule that is optimized for instruction cache locality. Furthermore, we identified a sweet spot where the ratio of data locality and instruction locality produces the fastest schedules.

Future work could focus on the development of a general framework for transforming an arbitrary QEP into QTM. Furthermore, work sharing among concurrent queries are an interesting research area. Finally, a cost model that predicts the costs of different task configurations on different hardware architectures could extend our approach significantly.

Chapter 5

Counter-Based Query Analysis

Modern processors employ sophisticated techniques such as speculative or out-of-order execution to hide memory latencies and keep their pipelines fully utilized. However, these techniques introduce high complexity and variance to query processing. In particular, these techniques are transparent to DBMS operations since they are managed by processors internally. To fully utilize the sophisticated capabilities of modern CPUs, it is necessary to understand their characteristics and adjust operators as well as cost models accordingly.

In this chapter, we exploit the *Performance Monitoring Units (PMU)* in modern CPUs to learn characteristics of database operations. As a case study, we extensively examine the execution of a relational selection operator (called *selection* in the remainder) on modern hardware in an in-depth performance analysis. We demonstrate, that branching behavior and memory exploitation are two main contributors to run-time. Based on these insights, we show how two common cost models would predict execution costs and why they fall short in determining run-time behavior for parallel execution. We reveal, that cost models which exploit only one performance parameter to determine execution costs are not able to predict the non-linear performance characteristics of modern CPUs. In the next chapter, we will exploit this knowledge to optimize query execution.

The rest of this chapter is structured as follows. In Section 5.2, we show how selections are transformed into executable code. Then, we present our case study of a relation selection operator in Section 5.3. Next, we highlight three major performance factors that determine the performance of a selection. First, Section 5.4 introduces branch prediction and its induced misprediction penalty. Second, we analyze the number of induced cache accesses including their impact on the memory hierarchy in Section 5.5. Third, we examine prefetching in Section 5.6. Based on these performance analysis, we investigate how parallelization changes selection characteristics in Section 5.7. Finally, we show how common cost models predict these changing characteristics in Section 5.8 before concluding this chapter in Section 6.7.

We published the results of this chapter in [ZF15]. Furthermore, the results in Section 5.6 originated in cooperation with a student research project by Daniel Lunow.

5.1 Performance Counters for Databases

Today's processors implement many sophisticated features to accelerate the performance of general-purpose applications. These features are transparent to applications like DBMSs and their usage depends on internal processor states such as resource or memory bandwidth utilization. Research in the field of single-thread DBMS performance shows, that main performance contributors are correctness of speculative execution [Ros04], exploitation of out-of-order execution [Rea98], prefetching [Hea07b], utilization of the instruction pipeline [Aea99], and exploitation of the multi-level cache hierarchy [Aea99, Pir13, Bea99, Kea98]. Another field of DBMS performance research focuses on the aspect of parallel query execution. Research in this area examines the exploitation of hyper-threading [ZCRS05], multiple cores for join operations [Bea13], and multiple sockets [Lea14] to parallelize query execution.

Performance analyses in both research areas could be divided into two categories. Studies in the first category examine DBMS workloads based on overall run-time. The general assumption is, faster execution methods utilize hardware resources more efficiently. Studies in this category mainly investigate the relational join as one of the most complex and time consuming database operation [Bea13, Kea12, SYT93, LR05]. In contrast, studies in the second category analyze DBMS workloads based on performance counters [Aea99, Kea98, Hea07b]. Using performance counters, they measure the efficiency of different CPU components such as branch prediction or pipeline utilization as well as the exploitation of the multi-level cache hierarchy. Studies in this category mainly investigate entire DBMS workloads such as different TPC benchmarks to show, how efficiently a DBMS exploits its available resources.

Studies that utilize performance counters exhibit several shortcomings. At first, several studies are performed on CPU simulators instead of real processors [Rea98, Kea98]. A simulator enables processor configurations which are most probably not available in any existing CPU. Second, these studies were conducted 15 years ago and thus rely on outdated processors. For example, they do not take multiprocessor technology into account. Third, row-oriented data layouts are examined instead of today's commonly used column-oriented data layouts. Finally, these studies run OLTP or OLAP workloads on commercial DBMSs to infer their exploitation of hardware resources. In these workloads, multiple operators interfere with each other during execution and thus characteristics of individual operators could be

misinterpreted. Furthermore, commercial DBMSs introduce several layers of complexity for logging, locking and other maintenance tasks which could potentially distort a single operator analysis.

In this chapter, we argue, that complex workloads on commercial DBMSs do not reveal the performance characteristics of individual operators. Therefore, we isolate the relational selection operator as a basic building block in complex DBMS workloads and execute micro-benchmarks to analyze its performance characteristics on modern processors. Because selections are commonly pushed down in the execution plan and thus are applied to many tuples, performance characteristics of selections are very important for overall query execution time. We will show, how processor features like branch prediction or multi-level cache hierarchies impact selection performance, especially for parallel execution. By sampling sequential and parallel selection execution, we reveal their different run-time characteristics.

Previous work sampled commercial DBMS workloads to identify the distribution between time spent for computation and time spent for waiting on data [Kea98, Rea98, TGA13]. Ailamaki et al. [Aea99] examined four major commercial database systems. They discovered, that on average, half of the execution time is spent in stalls while 90% of the memory stalls are due to L2 data cache misses and L1 instruction cache misses. Other research show similar distributions [Kea98, Rea98]. Tözün et al. [TGA13] point out, that L1 instruction cache misses have deeper impact than data cache misses for OLTP workloads. However, most studies use old CPU architectures with only two cache levels [Aea99]. Additionally, they sample query execution of the entire DBMS and do not examine the effects of parallelization and prefetching. In contrast, we analyze micro-benchmarks on the latest four Intel micro-architectures to identify the characteristics of a selection. Our results are independent of a particular DBMS implementation. Additionally, our time distributions on new CPU architectures differ greatly compared to the over 15 years old CPUs used in previous studies.

In the context of different scan variants, Broneske et al. [Bro15] and Răducanu et al. [RBZ13] examine different implementations of a scan operator. They showed, that the best variant depends on parameters such as selectivity, data distribution, and processor architecture. Additionally, some approaches exploit SIMD to accelerate scans [Wil09] or introduce bit-parallelism [LP13]. However, these approaches compare different variants only by run-time. In contrast, we examine a basic implementation and reveal its efficiency on current CPU architectures. Furthermore, we use performance counter to reveal which CPU component contributes most to the consumed run-time.

5.2 Background

In the remainder of this chapter, we use the following SQL query as a running example to analyze performance characteristics of a selection:

```
Select Sum(B) From tab Where A <= selValue
```

Our in-memory data set consists of 10M synthetically generated, randomized integer values in two column-oriented arrays (A and B). We adjust selectivity based on `selValue`. In Table 5.1, we present our test environment that contains four different CPUs based on Intel's latest micro-architectures. If not stated otherwise, we present test results on CPU 1. If they differ from CPU 2-4, we present them explicitly.

	CPU 1	CPU 2	CPU 3	CPU 4
Type	Intel Xeon	Intel Core i7 Mobile	Intel Xeon	Intel Core i7 Mobile
Model	E5-2630 v2	i7-2640M	E7-4870	i7-4900MQ
Microarchitecture	Ivy Bridge	Sandy Bridge	Nehalem	Haswell
Fequency	2.6 GHz	2.8 GHz	2.4 GHz	2.8 GHz
Physical Cores	6	2	10	4
L1 Instruction Cache	6x32KB 8-way	2x32KB 8-way	10x32KB 4-way	4x32KB 8-way
L1 Data Cache	6x32KB 8-way	2x32KB 8-way	10x32KB 8-way	4x32KB 8-way
L2 Unified Cache	6x256KB 8-way	2x256KB 8-way	10x256KB 8-way	4x256KB 8-way
L3 Unified Cache	15MB 20-way	4MB 16-way	30MB 24-way	8MB 16-way

Table 5.1: Test Systems.

Our example SQL query can be transformed into the following C++ code (assuming column-oriented data layout):

```
for (int i = 0; i < tupleCnt; i++)
   if(A[i] <= selValue)
         sum += B[i];
```

This C++ code iterates over all elements in the data set. For each tuple, it first checks if its attribute value $A[i]$ is less or equal to the current selection value. If $tuple_i$ qualifies, its attribute value $B[i]$ is added to the overall sum. Thus, the predicate evaluation is implemented as a conditional *if* statement in C++.

As a final step, a compiler translates C++ code into machine instructions. For each selection, the compiler generates one comparison instruction followed by a conditional jump instruction. Additionally, one such pair and a loop counter is generated for the entire loop. The conditional jump instruction determines the execution path as follows. The branch/jump is *not taken* and thus the execution continues with the next instruction if $tuple_i$ qualifies the selection predicate. On the other hand, a branch/jump is *taken* and thus the program execution jumps to the end of the loop code to test the loop condition if $tuple_i$ does not qualify.

```
1 LOOP_START:
2     cmpq $rcx,(%rax,%rdx,1) ;compare A[i] to selVal
3     ja LOOP_END ;jump if above (!qualified)
4     add (%rbx,%rdx,1),%r12; sum+= B[i]
5 LOOP_END:
6     add $0x8,%rdx; array offset (i) += 8
7     cmp $x320,%rdx ;compare i to loop counter
8     jne LOOP_START ;jump to LoopStart if !=
```

Listing 5.1: Assembler Code

Listing 5.1 (compiled with gcc 4.9) shows a simplified assembler implementation of our example query using two comparison instructions, two conditional branches, and two arithmetic additions. In a prolog (not shown), start addresses of the input arrays and a `selValue` are loaded into registers and a loop offset and temp sum variable are initialized. Registers are preloaded with the following values: `rax` = *start of array A*, `rbx` = *start of array B*, `rcx` = *selValue*, `rdx` = *array offset i*, and `r12` = *temporal sum*. For better readability, we omit physical addresses and introduce `LOOP_START` and `LOOP_END` as jumping labels. In Line 2, $A[i]$ is loaded and compared to `selValue` (in `rcx`). The `rax` register specifies the start address of column A and `rdx` stores the current array offset which represents the loop counter variable `i` in the C++ code. In Line 3, the outcome of the previous comparison is evaluated. The execution jumps to the end of the loop (to Line 5) if $A[i]$ is greater than `selValue` and thus $tuple_i$ does not qualify. Otherwise, the execution is continued in Line 4 by adding $B[i]$ to the overall sum in register `r12`. Note, this instruction is only executed if $tuple_i$ qualifies.

In Listing 5.1, loop counter `rdx` is simultaneously used as an array offset. Therefore, we increment `rdx` by 8 (size of one tuple) in Line 6 to prepare the next iteration. In Line 7, the new offset is compared to the number of required iterations. Listing 5.1 assumes 100 tuples and thus the loop terminates if the offset is equal to 800 ($hex = x320$). Finally, in Line 8, the execution jumps either to the beginning of the loop (Line 1) if another iteration is required, or otherwise leaves this code fragment by terminating the loop.

5.3 Case Study Selection

In this chapter, we isolate the relational selection operator and analyze its performance in-depth using micro-benchmarks. In Figure 5.1, we present run-time characteristics of a selection using different degrees of parallelism (DOP). We show *CPU time* in *cycles per input tuple* on the y-axis and selectivity on the x-axis. CPU time measures the total time spent by all CPUs individually instead of execution time (so-called wall-clock time). By ana-

lyzing these results, we reveal three important performance characteristics of a selection. First, a selection does not scale linearly with the number of cores. A linear scaling would be indicated by the same CPU time but less wall-clock time. Second, curves change their trends from a peak at 50% selectivity with two declining edges (DOP 1) to a sharp increase followed by constant pathway (DOP 24). Between these extremes, there are several transitional curves. Third, common cost models, i. e., Ross [Ros04] and Pirk et al. [Pir13], approximate correct execution costs only for a subset of the entire DOP range. We will show, that these cost models are insufficient to predict the selection performance on modern processors.

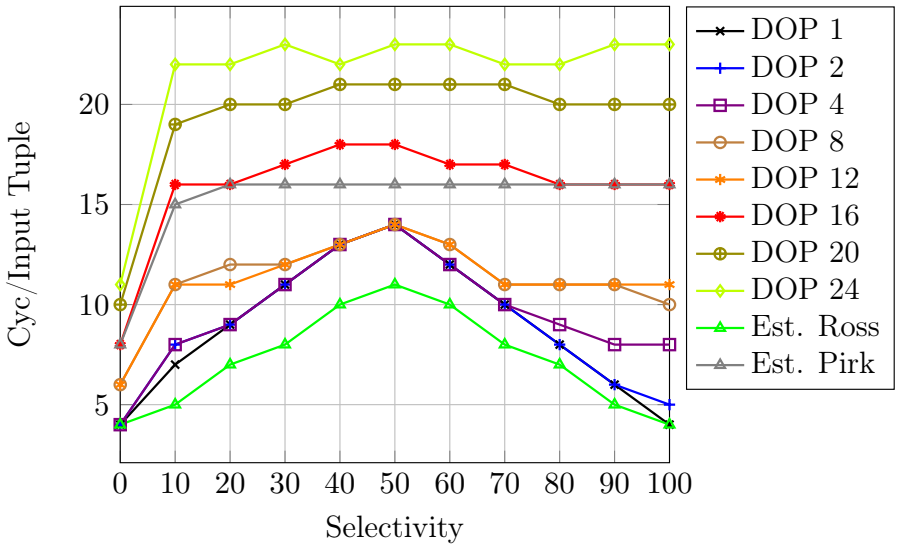


Figure 5.1: Selection Scalability.

In summary, our analysis reveals the following insights:

- The branch prediction algorithms in modern Intel CPUs did not change among the latest four micro-architectures.
- The number of branch misprediction are deterministic and predictable.
- Different prefetchers available in modern CPUs largely impact the performance but they are predictable too.
- Parallelization changes the run-time behavior of selection completely and common cost models fall short for parallelization.
- Run-times of selections with small DOPs are determined by the branching behavior. In contrast, selection using a large DOP are bound by cache accesses.

5.4 Branch Prediction

Research by Ross [Ros04] showed, that branch prediction is one major performance contributor for a selection. A branch predictor in modern CPUs has two alternative options to predict the outcome of a branch. The correctness of its prediction is essentially to utilize processor pipelines efficiently. First, *static* branch prediction determines that forward jumps, e. g., a if statement, are not taken and backward jumps, e. g., at the end of a loop, are taken. This simple prediction scheme is applied if no other information is accessible for a branch [Int12b]. Second, *dynamic* branch prediction determines the outcome of a branch based on its branch history. A *Branch Target Buffer (BTB)* saves the branch address as well as its last outcomes to recognize patterns of branches taken/not taken.

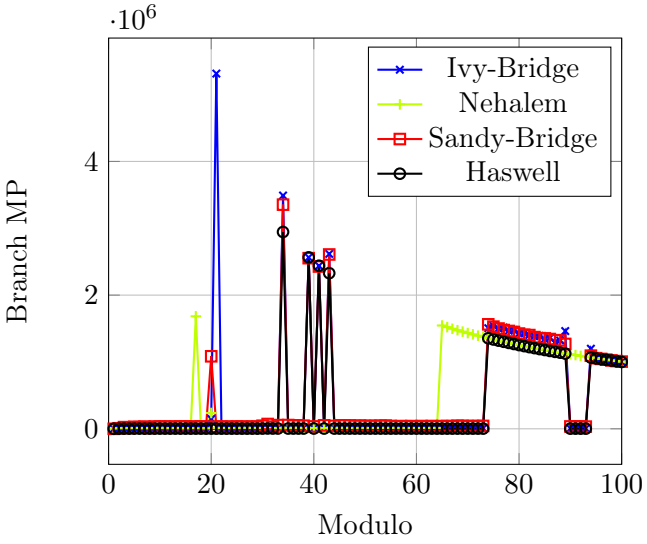


Figure 5.2: Branch History Buffer.

To examine the pattern size that modern CPUs recognize, we create different patterns based on a modulo division (if($value_{1...n}$%x)). In Figure 5.2, we vary the modulo values from one (all tuples qualify and thus all branches are not taken) to 100 (only each 100th branch is not taken). Thus, x indicates that only each x-th branch is not taken and all other branches are taken. If the pattern is not recognized by a CPU, each x-th branch will be mispredicted (plotted on the y-axis). As shown, Ivy-Bridge, Sandy-Bridge, and Haswell CPUs (CPU 1, 2, 4 in Table 5.1) detect patterns with up to 72 different outcomes. Starting from $x = 72$, each *branch not taken* induces one branch misprediction and thus we deduce that these patterns are too long to be recognized. Because a BTB stores patterns in a circular manner [Int12b], starting from $x = 72$, each additional outcome overwrites an existing entry. Nehalem as the oldest micro-architecture exhibits a smaller BTB and is ca-

pable of detecting patterns up to 64 different outcomes (CPU 3 in Table 5.1). We emphasize that we cannot point out a particular reason for spikes around $x = 20$ and $x = 40$ in Figure 5.2 as well as improved branch prediction between $x = 90$ and $x = 93$. Although they are reproducible, information found in the Intel manual does not explain their occurrence [Int12b]. As a result, we conclude that modern CPUs recognize branching patterns of selections that repeat their history within less than 72 consecutive outcomes. These patterns are mostly introduced by predicates with very high or very low selectivities and explain their excellent right prediction rate.

In general, the branching pattern of a selection is determined by its selectivity p. Following Ross [Ros04], we assume a processor with a *perfect* branch predictor. For a selectivity below 50%, it predicts that each tuple does not qualify and thus each branch will be taken. On the other hand, for a selectivity above 50%, it predicts that each tuple qualifies and thus each branch will not be taken. Because the number of output tuples is equal to the number of branches not taken (BNT), we calculate the number of branch mispredictions:

$$BRMP(p) = \begin{cases} BNT(p), \text{if p} \leq 0.5 \\ BNT(1-p), \text{if p} > 0.5 \end{cases} \tag{5.1}$$

Thus, for a selection with a selectivity below 50%, the branch predictor predicts that each tuple does not qualify (branch is taken) and therefore *mispredicts* each qualifying tuple (branch not taken). Hence, the number of branch mispredictions is equal to the number of branches not taken. On the other hand, for a selection with a selectivity above 50%, the branch predictor predicts that each tuple qualifies (branch is not taken) and thus mispredicts each not qualifying tuple (branch taken). Based on the number of mispredictions and the number of conditional branches (taken + not taken branches), we calculate the number of right predictions:

$$BRRP(p) = Conditional\ Branches - BRMP(p) \tag{5.2}$$

Note that, a loop itself induces as many branches as input tuple exists. However, these branches are almost always taken and thus correctly predicted (except for the last iteration).

In Figure 5.3, we evaluate Equation 6.4 on the latest four Intel micro-architectures: Nehalem, Sandy-Bridge, Ivy-Bridge, and Haswell. As shown, the estimated number of branch mispredictions matches the measured branch mispredictions for all micro-architectures. However, around 50% selectivity, CPUs mispredict slightly more branches than Equation 6.4 estimates. Additionally, branch prediction on Nehalem deviates more from Equation 6.4 compared to Intel's latest three micro-architectures. Following Ross [Ros04], we could estimate branch-induced costs for a selection by combining the estimated mispredictions with a penalty.

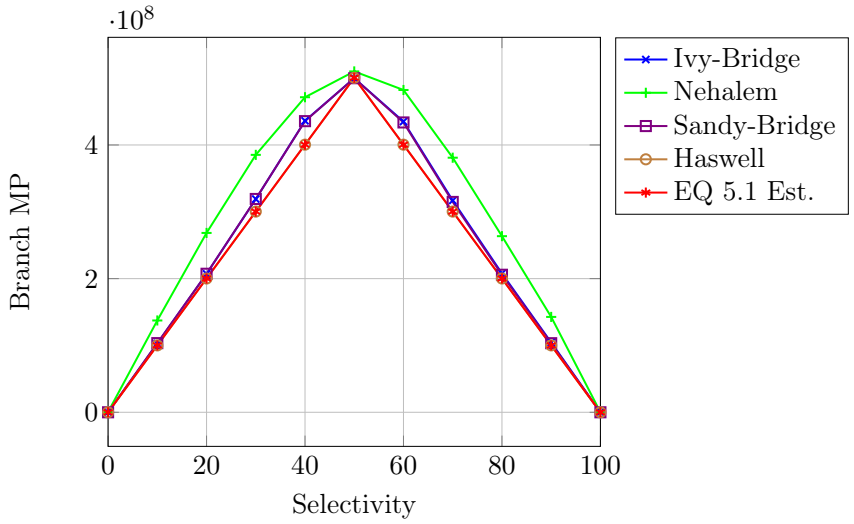


Figure 5.3: Branch Misprediction.

Figure 5.4 summarizes relationships between branch-related counters for a selection. First, the number of conditional branches are constant for the entire selectivity range. Second, branches not taken (BNT) and branches taken (BT) converge with increasing selectivity. Because the number of conditional branches remains constant, each additional qualifying tuple reduces the number of not qualifying tuples by one. At zero percent selectivity, no tuple qualifies and thus the branch is taken for each tuple. Additionally, each loop iteration (back to the loop start) induces one BT; thus, the number of branches taken are twice as big as the number of input tuples. For a selectivity of 100%, each tuple qualifies and thus each branch is not taken by the predicate evaluation. Additionally, one branch is taken for each loop iteration.

In contrast, branch prediction shows a different trend. For selectivities below 50%, each not qualifying tuple (BT) results in a right branch prediction and each qualifying tuple (BTN) in a branch misprediction (indicated by the overlapping lines). In contrast, for selectivities above 50%, this correlation switches such that each qualifying tuple result in a right prediction and each not qualifying tuple in a branch misprediction. Thus, a predictable branching behavior (few mispredictions) are induced by very high or very low selectivities. It is important to note, that the number of branches taken and not taken are processor-independent because their occurrences are determined by the input data and the predicate. In contrast, their prediction depends on the CPU internal branch prediction algorithm. Figure 5.3 reveals, that these branch prediction algorithms did not change among the latest four Intel micro-architectures. Therefore, branch-related behavior of a selection on modern Intel CPUs is deterministic and can be approximated.

In sum, the branching behavior of modern CPUs impact the performance of a selection significantly. However, branch prediction follows a predictable and consistent pattern. We can utilize performance counters to measure branching events. Based on the results, we can make assumptions about the data that are processed by a selection.

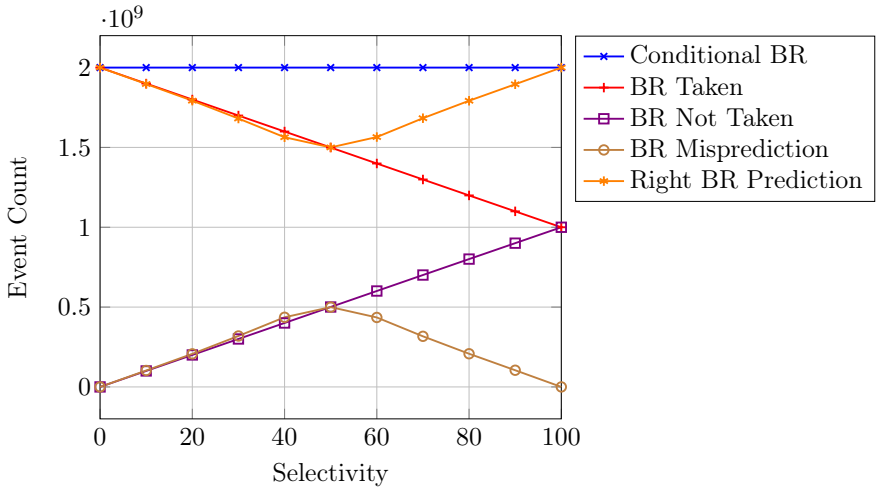
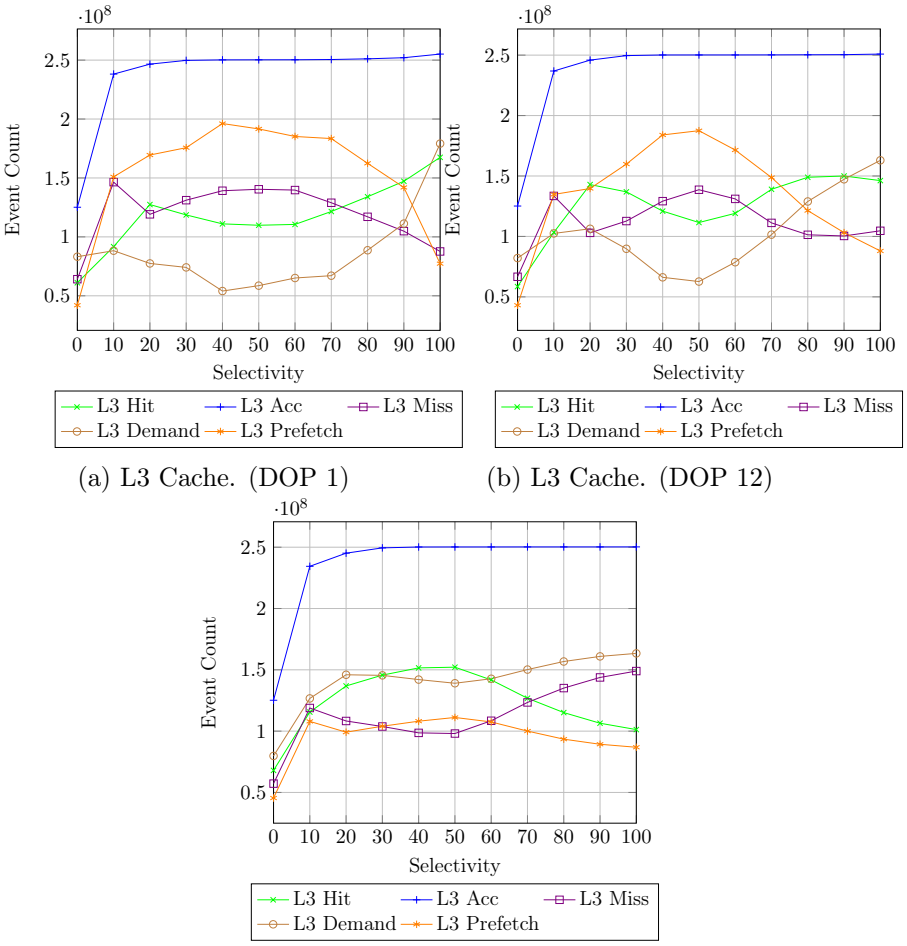


Figure 5.4: Branch-related Counter.

5.5 Cache Misses

Cache accesses and their induced cache misses are the second the major performance contributor of a selection. The extension of the generic cost model (see Manegold et al. [Man02]) by Pirk et al. [Pir13] allows us to model cache accesses for a selection by combining two access patterns. First, a selection introduces a *sequential traversal* access pattern that in turn induces one random cache miss for accessing the first cache line and one sequential miss of each subsequent cache line. Second, each subsequent selection introduces a *sequential traversal with conditional reads* access pattern which induces cache accesses depending on the selectivity of the previous selection. In our example query, the aggregation function conditionally accesses column B only for tuples that qualify on column A. A selectivity of zero percent represents a baseline for accessing only column A. For increasing selectivity, additional cache line accesses to column B and branch misprediction penalties are induced.

In Figure 5.5a, we plot L3 cache-related performance counters for our example query using a DOP of one. We exclude L1 and L2 counter values from Figure 5.5a because they show similar values compared to the L3 cache. The main reasons are the streaming access pattern (without tuple reuse)



(a) L3 Cache. (DOP 1)

(b) L3 Cache. (DOP 12)

(c) L3 Cache. (DOP 24)

Figure 5.5: L3 Cache Overview.

and the inclusive property of the L3 cache. We will discuss Figure 5.5b and Figure 5.5c in Section 5.7.

In Figure 5.5a, L3 cache accesses increase up to a selectivity of 20% and then remain constant. The cost model by Pirk et al. [Pir13] estimates this trend by considering the probability of a cache line access. In the selectivity range from 0% to 20%, some cache lines are not accessed and thus random memory accesses occur. With increasing selectivity up to 20%, the probability that two memory references access the same cache line increases. For a selectivity larger than 20%, each cache line is accessed and thus the number of cache accesses remain constant among the entire selectivity range from 20% to 100%. Note that, the actual switch point depends on the number of tuples per cache line [Pir13].

L3 cache accesses are composed of demand accesses (created by load instructions) and prefetch accesses (created by CPU prefetchers). As shown in Figure 5.5a, demand accesses are induced more frequently for low and high selectivities. Towards a selectivity of 50%, they decrease and increase thereafter (indicated by a dip). In contrast, prefetch accesses show an opposite trend. For high and low selectivities, less prefetches are induced by CPU prefetching units. Towards a selectivity of 50%, most prefetches are induced with falling edges to both sides. The main reason for this trend is the branch prediction which shows the same characteristics as the prefetch accesses. At 50% selectivity, most branches are mispredicted and thus many unnecessary instructions and data loads for not taken execution paths are induced. Thus, prefetchers trigger more often and the number of demand accesses decreases.

In general, a demand or prefetch cache line request can either be a cache hit or a cache miss. The ratio between hits and misses depends on the temporal gap between demand access and prefetch accesses as well as the branch prediction correctness. At first, a prefetch from a mispredicted execution path induces one cache miss because its cache line is never used. In contrast, two cache misses occur if a prefetch of a useful cache line is issued either too early (evicted before used) or issued too late (not completed when accessed by demand); thus, memory bandwidth is wasted by prefetching. In the best case, a prefetcher requests a cache line in time such that the prefetch itself misses the L3 but the following demand access hits.

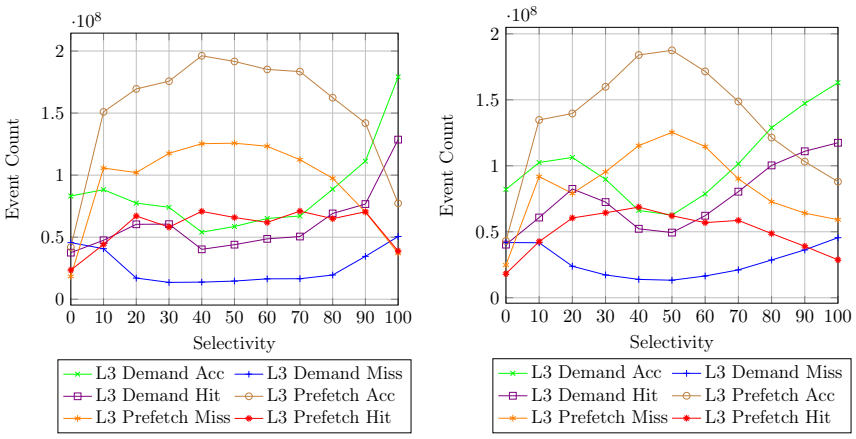
The cache hit and cache miss curves show also a contrary trends in Figure 5.5a with a switch point at 50% selectivity. Whereas L3 hits follow the trend of L3 demand accesses, L3 misses follow the trend of L3 prefetches. The prefetching units in modern CPUs produce these effects. In general, prefetches induce cache misses because they access tuples most probably at first. If prefetches are issued in time and from correct execution paths, only one L3 miss occurs. With an increasing number of branch mispredictions, the number of unused prefetches increase and thus the number of cache misses. On the other hand, issuing prefetches out of time is shown in Figure 5.5a by sequential access to column A (0% selectivity) and sequential access to columns A and B (100% selectivity). Although branches are predicted almost always correctly, L3 misses are still induced because the memory bandwidth is overexerted. However, the number of demand accesses and hits are also high.

In Figure 5.5a, all counters increase steeply for a selectivity between zero and ten percent. In this selectivity range, demand and prefetch accesses as well as cache hits and misses follow the steep increase in L3 accesses, which are in turn explained by the probability of a cache line access [Pir13]. The sharp increase of cache misses in this range follows a sharp increase in prefetching.

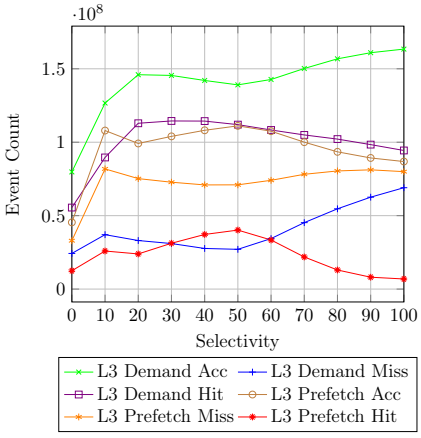
In Figure 5.6a, we show a detailed breakdown of demand-related L3 cache counters. As shown, demand misses are most frequent for low and high

selectivities. In contrast, demand accesses often hit the cache, especially in the medium-selectivity range. This observation explains why prefetches are issued only rarely in low and high selectivity ranges, but frequently in the medium-selectivity range. For high selectivities, the number of demand misses suddenly increases. In this case, the increased number of accesses shortens the amount of time between two accesses; thus, the prefetcher is not fast enough to prefetch each cache line access. Additionally, prefetchers decrease their efforts in this range.

Overall, prefetchers in modern CPUs perform very well for sequential access patterns which is indicated by the small gap between hits and accesses. To enable efficient prefetching, CPUs exploit two types of prefetchers [Int12b]. First, the L1 and the L2 *streaming prefetcher* fetch the next cache line. Second, the L1 and the L2 *stride prefetcher* exploits load histories to detect and prefetch strided forward or backward loads. For a selection, se-



(a) L3 Demand/Prefetch (DOP 1). (b) L3 Demand/Prefetch (DOP 12).



(c) L3 Demand/Prefetch (DOP 24).

Figure 5.6: L3 Demand and Prefetch.

quential access to column A induces a simple streaming pattern which is well suited for these prefetchers. However, access to column B induces irregular strides, especially for medium-selectivity ranges, which results in less efficient prefetching. In Section 5.7 we will show, that an increased number of prefetches for a DOP of one is less adverse compared to larger DOPs.

Finally, Figure 5.6a shows a detailed breakdown of prefetch-related L3 cache counters. Surprisingly, not each prefetch access results in a prefetch miss as one might expect. Prefetch hits could be induced by different prefetchers. Thus, if two prefetchers prefetch the same cache line within a specific temporal gap, the first will miss but the second will be successful, i. e. prefetch hit. We emphasize that, L1 *line fill buffers* catch accesses to multiple tuples within the same cache line and forward only one load or prefetch request to lower cache levels (L2 and L3 cache) [Int12b]. As a result, the access to different tuples in the same cache line will not lead to prefetch hits.

5.6 Prefetching

Intel CPUs enable users to deactivate individual prefetcher using MSR registers [Int17a]. In the following, we use our selection example from Section 5.2 in combination with different prefetchers disabled to derive the impact and functionality of different prefetchers. In Figure 5.7, we contrast the execution of a selection with all prefetcher enabled (see Figure 5.7a) and all prefetchers disabled (see Figure 5.7b). At first, a selection without prefetching induces an up to two times longer run-time compared to a selection using prefetches. Second, the performance gap between DOP 1 and DOP 12 is larger for medium selectivities and smaller for high selectivities. Third, for a DOP 1 and 12, the trend of the lines in Figure 5.7 remain similar. In contrast, for a DOP of 24, the worst-case performance of a selection without prefetching shifts to 30% selectivity and induces a less sharp transition compared to a selection using prefetching. In Section 5.6.1 and Section 5.6.2, we examine the L1 and L2 prefetchers in detail to explain this behavior.

5.6.1 L1D Hardware Prefetchers

Modern Intel CPUs consist of two L1 prefetchers which prefetch cache lines into the L1 data cache (L1D) [Int12a]:

- *Data cache unit (DCU) prefetcher.* This prefetcher, also known as the streaming prefetcher, is triggered by an ascending access to very recently loaded data. The processor assumes that this access is part of a streaming algorithm and automatically fetches the next line.
- *Instruction pointer (IP)-based stride prefetcher.* This prefetcher keeps track of individual load instructions. If a load instruction has a regular

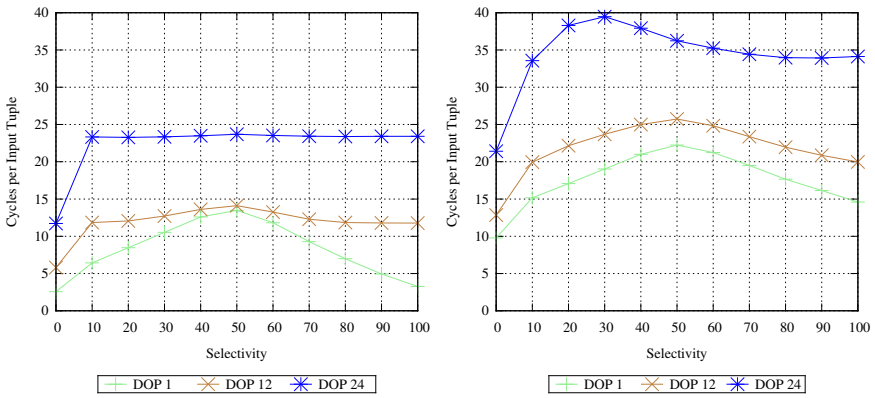
stride, then a prefetch is sent to the next address which is the sum of the current address and the stride. This prefetcher can prefetch forward and backward and can detect strides of up to 2 KB.

The DCU prefetcher only prefetches the next cache line. In contrast, the IP-based stride prefetcher can prefetch 2 KB ahead. Furthermore, the DCU prefetcher detects sequential access patterns across different load instructions whereas the IP-based stride prefetcher considers only loads issued by the same load instruction for striding access.

A prefetch is triggered by a load instruction if the following conditions are satisfied [Int12a]:

1. The load instruction is from a write back memory type.
2. Prefetched data is on the same 4 KB page.
3. No fence is in progress in the pipeline.
4. Few other load misses are in progress.
5. There is no continuous storage stream.

The first condition restricts prefetching to main-memory. In contrast, a write-through memory type is mainly used for device drivers and might bypass caching completely. The second condition restricts prefetching within the same 4 KB page and is required to avoid page translation. A page translation obtains the physical address of a page if this translation is not already stored in the TLB cache by a previous access. To prevent this time-consuming translation, a prefetcher only prefetches cache lines on pages with addresses already stored in the TLB cache. The third condition prohibits prefetches while fences are in progress. A fence instruction ensures a global order of memory operations by temporally partitioning memory operations



(a) Hardware Prefetching enabled. (b) Hardware Prefetching disabled.

Figure 5.7: Selection with and without prefetching.

which are executed before and after the fence. By disabling prefetching during fence operations, fences could be implemented more efficiently. The fourth condition introduces a quantitative assumption. If *many* load misses are in progress, a CPU might infer that the load pattern is most probably irregular. Otherwise, the loads would hit the L1 cache. In case of an irregular access pattern, further prefetches could pollute the available loading slots which are used by demand accesses too. Furthermore, induced prefetches of unused cache lines could evict useful cache lines. The fifth condition restricts prefetching if a sequence of store operations is in progress. In general, a cache line has to be accessed before it could be written. Thus, by disabling prefetching, the memory bus is freed-up from unnecessary prefetching. To summarize, the first three restrictions are imposed by architectural considerations while the last two restrictions aim to prevent prefetcher-related performance degradation.

In Figure 5.8, we present demand accesses (green line), prefetch accesses (brown line), demand misses (blue line), and demand hits (red square). Additionally, we divide prefetches into prefetches issued by the L1 IP-based stride prefetcher (purple square) and prefetches issued by the L1 DCU prefetcher (red circle). We gather these counters by disabling either the IP-based or the DCU prefetcher. As shown in Figure 5.8, the numbers of prefetches by the DCU and IP-based prefetcher do not add up to the total number of prefetches. The main reason for that is that prefetchers might behave differently and thus exploit the prefetch capacity differently if other prefetchers are disabled or enabled. In particular, individual prefetchers might utilize the request buffers between the first-level cache and the second-level cache differently [Int12b].

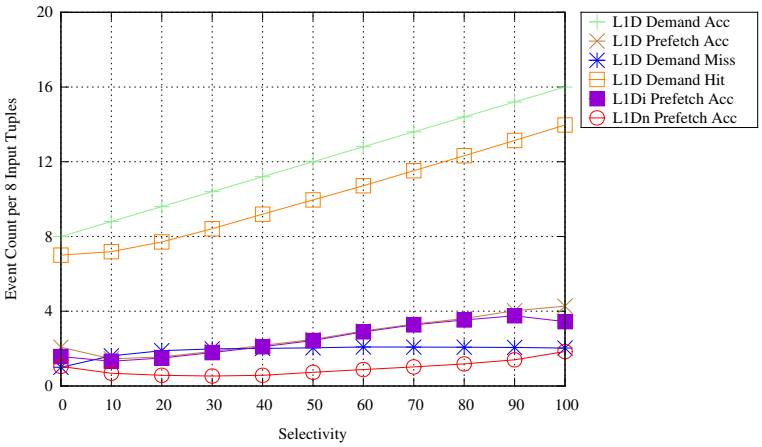


Figure 5.8: L1D Accesses at DOP 1.

Figure 5.8 shows that the IP-based stride prefetcher issues more prefetches than the DCU prefetcher across all selectivities. In particular, the IP-based

stride prefetcher is responsible for most of the prefetches because it curve overlay the prefetch accesses with all prefetcher enabled. Additionally, both prefetchers increase their prefetching efforts for larger selectivities; thus, if more tuples qualify. This effect confirms the condition that prefetcher dynamically regulate their effort based on the number of load misses. For our example selection query (see Section 5.2), the IP-based stride prefetcher detects the sequential access pattern to the first array. In contrast, the random conditional access to the second array triggers the IP-based prefetcher irregularly. Finally, the latency to main memory is not sufficient to hide all cache misses because our example selection query contains a very tight loop.

5.6.2 L2 Hardware Prefetchers

There are two L2 hardware prefetchers which prefetch cache lines into the L2 and L3 cache. Both prefetchers prefetch data to the last level cache (LLC). Typically, data is brought also into the L2 cache unless the L2 cache is heavily loaded with missing demand requests. These L2 prefetchers are [Int12b]:

- *Spatial Prefetcher*: This prefetcher strives to complete every cache line fetched to the L2 cache with the pair line that completes it to a 128-byte aligned chunk.

- *Streamer*: This prefetcher monitors read requests from the L1 cache for ascending and descending sequences of addresses. Monitored read requests include L1 DCache requests initiated by load and store operations and by the hardware prefetchers. When a forward or backward stream of requests is detected, the anticipated cache lines are prefetched. Prefetched cache lines must be in the same 4 KB page.

The latest Intel architecture CPUs implement the following enhancements for the *streamer* [Int12b]:

- The streamer may issue two prefetch requests on every L2 lookup. The streamer can run up to 20 lines ahead of the load request.

- The streamer adjusts dynamically to the number of outstanding requests per core. If there are only few outstanding requests, the streamer prefetches further ahead. If there are many outstanding requests or the prefetched cache line is far ahead, the streamer prefetches to only into LLC and shorten its prefetch distance.

- The streamer detects and maintains up to 32 streams of data accesses. For each 4 KB page, one forward and one backward stream can be maintained.

The spatial prefetcher follows the same idea as the L1 DCU prefetcher which fetches the next cache line based on the currently loaded address. The streamer follows the same idea as the IP-based L1 prefetcher which monitors load instruction to detect pattern and eventually prefetch anticipated cache lines. Both L2 hardware prefetchers dynamically adjust themselves by prefetching only to the L3 cache if many demand requests miss the L2 cache. Additionally, the streamer prefetches only to the LLC cache if it runs far ahead of the current demand request. However, if a demand requests accesses cache lines which are only present in the LCC cache, the streamer can issue an additional prefetch to bring these cache lines into the L2 cache.

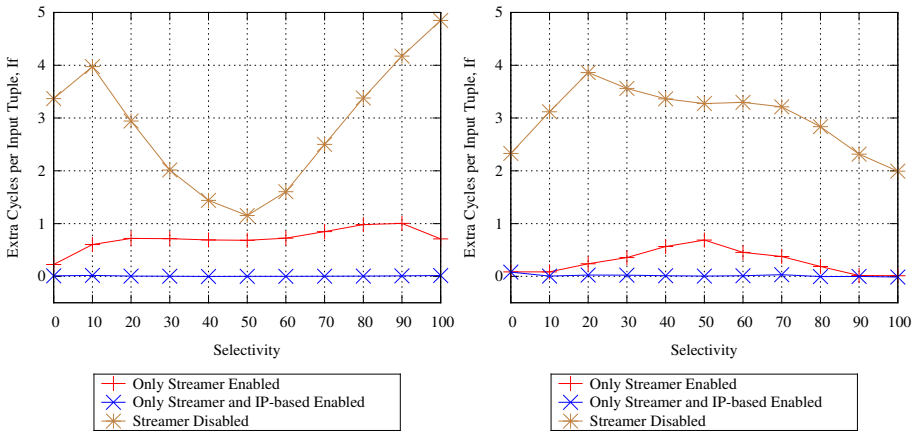
Figure 5.9 shows the performance penalty in cycles per input tuple for our selection example query using DOPs 1, 12, and 24 while different prefetchers are disabled. First, we disable all hardware prefetchers except the streamer (red line). As shown, the performance decreases only minor with up to one cycle per input tuple. Thus, we infer that the streamer has the most impact on the performance of a selection on modern CPUs. Second, if the streamer is disabled (brown line), the performance degeneration is significant. This observation supports the assumption that the streamer is the most important prefetcher for a selection on modern CPUs. Furthermore, the behavior of prefetching changes for different DOPs. For a DOP of one, prefetching is the inverse of to branch misprediction. In contrast, for higher DOPs, the penalty follows the number of L3 hits (see Figure 5.5c). We will discuss this trends in detail in Section 5.7.3. Third, by only enabling the streamer and the IP-based stride prefetcher, the performance is almost identical to the run-time where all prefetchers are enabled. We infer, that the IP-based prefetcher brings prefetches which only reside in the L2 cache into the L1 cache and thus supports the streamer. As a result, the spatial prefetching approach of fetching the next cache line as well as the L1 prefetchers are only minor important for a selection on modern CPUs.

5.7 Parallel Execution

In Section 5.4 and Section 5.5, we focused on sequential execution of a selection on one core and show how branch-related and cache-related performance counters change their characteristics. In Section 5.7.1, we examine these characteristics for parallel execution. After that, we present a time distribution of cycles spent in different CPU components in Section 5.7.2. Then, Section 5.7.3 relates different run-time characteristics to performance counters before we investigate selection scalability in Section 5.7.4.

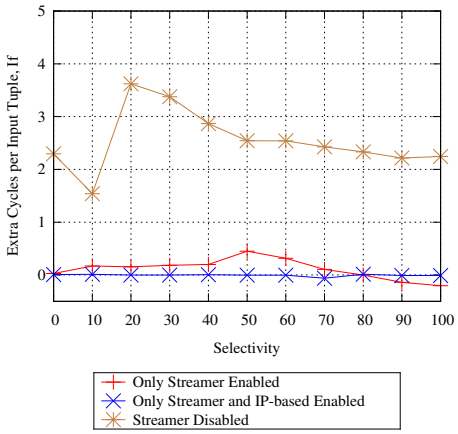
5.7.1 Degree of Parallelism

As shown in Sections 5.4 and 5.5, branch prediction and cache accesses are the major contributors to selection performance. For parallel execu-



(a) DOP 1.

(b) DOP 12.



(c) DOP 24.

Figure 5.9: Cache Misses.

tion, branch-related counters which reflect the branching behavior do not depend on the number of CPUs involved in the processing because they are *instruction-dependent*. If a selection is partitioned among multiple cores and executed in parallel, the number of conditional branches, branches taken, and branches not taken do not change. Instead, branches are distributed among partitions and their sum remains equal for a selection using one or multiple cores. Instead, the branching behavior of a selection depends on the selectivity and parallelism represents an orthogonal parameter.

In contrast, cache-related counters which reflect the memory utilization depend on the number of CPUs involved in the query processing because they are sensitive to *memory-bandwidth*. Figure 5.5 shows L3 cache-related counters for different DOPs. Note that, CPU 1 has 12 physical and 24 logical cores. Thus, starting from a DOP of 12, hyper-threading is applied. In

Figure 5.5, only L3 accesses remain constant among different DOPs. With increasing parallelism, three cache-related characteristics change. First, demand and prefetch accesses converge to each other. Second, more demand and less prefetch accesses are induced. Third, the correlation between hits and demand accesses as well as misses and prefetch accesses merge such that they partially overlap.

Figure 5.6 splits up demand and prefetch accesses as well as their induced misses and hits. Among all DOPs, prefetching works well such that the majority of demand accesses hit the L3 cache. However, demand accesses and hits change their trends as well as their occurrence if parallelism is applied. For small to medium DOPs (1 and 12 cores), less demand accesses are induced because prefetchers work more efficiently and thus cache accesses can be satisfied on higher cache levels. Typically, cache lines are brought to L2 cache unless it is *heavily* loaded with missing demand requests. As shown in Section 5.6, prefetchers in modern CPUs are sensitive to the overall memory bandwidth and thus the number of prefetches decrease with higher memory bandwidth utilization [Int12b]. Therefore, prefetching for low to medium DOPs work more efficient because they require less memory bandwidth. In particular, prefetchers increase their prefetching efforts in the medium-selectivity range for low to medium DOPs because memory bandwidth is available. In contrast, selections using 24 cores overexert memory bandwidth and thus less prefetches are induced. The reduced number of prefetches combined with a longer prefetching latency induced by the memory bottleneck result in more demand accesses.

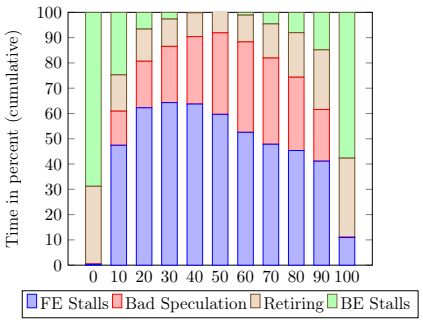
5.7.2 Time Distribution

In this section, we derive a time distribution for different CPU components following the *Intel optimization guide* [Int12a]. Intel provides special counters to monitor buffers that feed micro-ops supplied by the front end to the out-of-order back end. Using these counter, we are able to derive which CPU component stalls the pipeline for how long.

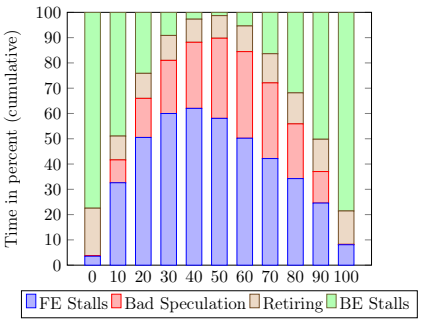
For our example query, we plot a time distribution of cycles spent in four CPU components based on the *Intel optimization guide* [Int12a] in Figure 5.10. First, the *front end* delivers up to four micro-ops per cycle to the back end. If the front end stalls, the rename/allocate part of the out-of-order engine will starve and thus execution becomes *front end bound*. Second, the *back end* processes instructions issued by the front. If the back end stalls because all processing resources are occupied, the execution becomes *back end bound*. Third, with *bad speculation*, the pipeline executes speculative micro-ops that never successfully retire. This component represents the amount of work wasted by branch mispredictions. Fourth, *Retiring* refers to the amount of cycles that are actually used to execute useful instructions. This component represents the amount of useful work performed by a processor.

Back end stalls can be further splitted into memory-related stall time and core-related stall time. Memory-related stall time corresponds to stalls related to the entire memory subsystem, e. g., cache misses that may cause execution starvation. In contrast, core-related stall time originates from execution starvation or non-optimal execution ports utilization, e. g., long latency instructions may serialize execution [Int12b]. For our example query, the ratio between core stalls and memory stalls is determined by the ratio between front end and back end stalls. Figure 5.10 shows that front end stalls as well as core stalls predominate a selection using one core. In contrast, back end stalls and memory stalls predominate a selection using all logical cores.

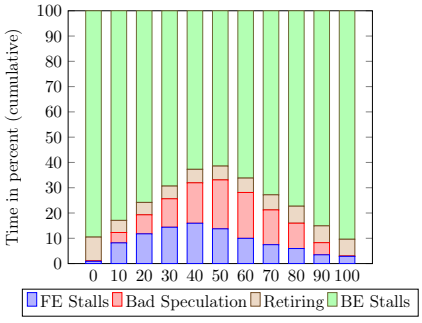
In Figure 5.10, a selection using a DOP of one and small selectivities spent the majority of time in the back end and for retiring the useful results. Thus, the processor is efficiently utilized and the memory bandwidth constitutes the limiting factor. For medium selectivities, the majority of stall time shifted towards bad speculation and front end stalls. Thus, a selection becomes front end bound. In general, bad speculation leads to a significant amount of wasted cycles and prevent instructions from entering the pipeline at the front end (front end pollution) [Int12b]. Figure 5.10 shows this relation by the correlation between bad speculation and front end stalls. For very large selectivities, bad speculation decreases in favor for more back end stalls



(a) Time Distribution for DOP 1.



(b) Time Distribution for DOP 12.



(c) Time Distribution for DOP 24.

Figure 5.10: Time Distribution.

and the time spent for useful computation (retiring) increases. These characteristics are similar to a very low selectivity. A detailed back end analysis for a selection using one core reveals, that the back end is dominated by the core time. Thus, the CPU uses the out-of-order execution engine inefficiently for medium-selectivities. In contrast, for very high and low selectivities, the back end time is dominated by memory stalls in the cache hierarchy. In sum, front end stalls and bad speculation prevail for a selection using one core and thus branch misprediction is the main contributor to the run-time.

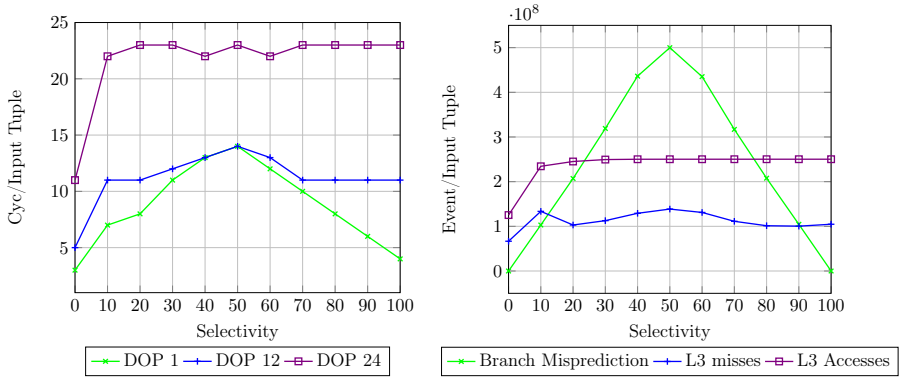
For a selection using all logical cores (24), the behavior changes completely. First, the overall time distribution shifted towards back end stalls. Second, the back end becomes predominated by memory stalls. However, the general trend for bad speculation and front end stalls remains with a peak at 50% selectivity but with a smaller portion of the overall time. As a result, back end stalls prevail for a selection using all logical cores and thus cache accesses mainly contribute to the run-time.

Finally, a selection using all physical cores (12) represents a middle ground between these both extremes and its main contributor to run-time depends on the selectivity. For low and high selectivity ranges (0% to 30 and 70% to 100%), the time spend in the back end increases compared to one core execution and thus cache accesses are more determining. In contrast, branch mispredictions prevail as the main contributors to run-time in the selectivity range from 40% to 60%.

5.7.3 Run-time Characteristics

Figure 5.11 presents run-times of our example query using a DOP of one, 12, and 24. As shown, run-time characteristics differ largely between these DOPs. For a DOP of one, run-time peaks at 50% with falling edges to both sides. In contrast, a selection executed by 24 logical cores exhibits a steep increase in run-time between zero and ten percent selectivity before staying constant among the remaining selectivity range up to 100%. Finally, a selectivity executed by 12 physical cores exhibits a middle ground between a DOP of one and 24. Therefore, it shows the same peak at 50% selectivity with falling edges to both sides, but passes over to constant pathways very sharply.

Our time distribution analysis in the previous section enables us to explain these different trends (see Section 5.7.2). In Figure 5.11b, we contrast run-times to performance counters which exhibit similar trends. For a DOP of one, run-time follows branch mispredictions. This run-time characteristic is in line with results presented by Ross [Ros04]. In contrast, run-time for a DOP of 24 follows L3 cache accesses. This run-time characteristic is in line with results presented by Pirk et al. [Pir13]. In between these two extremes, a selection executed by 12 physical cores follows L3 cache misses.



(a) Run-Time for different DOP. (b) Performance Counters.

Figure 5.11: Run-Time and related Performance Counters.

There are two main reasons for these changing run-time characteristics. First, as shown in Sections 5.4 and 5.5, branch mispredictions and L3 cache accesses are two major performance factors that determine the performance of a selection. If a selection is executed in parallel, each additional core reduces the effective bandwidth per core. We demonstrate in Figure 5.10a, that memory bandwidth is no limiting factor for a selection executed by one core because the bandwidth is not fully utilized. By disabling one major performance factor, the other has an increasing impact. Therefore, selections on one core are *branch prediction bound* and thus the number of introduced branch mispredictions determine the run-time. We confirm this observation in Figure 5.11b which shows that branch mispredictions exhibit the same trend as the run-time of a selection using one core. In contrast, a DOP of 24 overexerts memory bandwidth. Thus, a selection spent the majority of its cycles waiting on data transfers from memory (see Figure 5.10c). However, branch mispredictions still occur but they can be overlapped with memory accesses. Thus, they contribute only minor to the overall run-time such that a selection using all logical cores become *memory bound*. We confirm this observation in Figure 5.11b which shows that L3 cache accesses exhibit the same trend as the run-time of a selection using all logical cores.

Finally, a selection using all physical cores (12) spent less time waiting on data than a selection using all logical cores (24) (see Figure 5.10b). Because processor vendors commonly align memory bandwidth to the number of physical cores [Int12b], selections using only physical cores are more memory efficient. Thus, the curve in Figure 5.11b is composed of two intervals. In the first selectivity interval, from 0% to 30% and 70% to 100%, run-time is memory bound. In the second selectivity interval, from 40% to 60%, branch misprediction could not be entirely overlapped with memory accesses and thus the selection is branch prediction bound. However, L3 misses exhibit the same trend a selection using 12 physical cores and thus execution be-

come *cache miss bound*. In sum, selections shift their run-time characteristics from a branch prediction bound to a memory bound execution with several transitional trends.

5.7.4 Scalability

In Figure 5.12, we plot run-time speed-up for a selection using different DOPs compared to a DOP of one. Selections using two cores show a linear speed-up. Starting from a DOP of four, the speed-up changes among the entire selectivity range. For example, a selection using four cores scales only linear in the selectivity range from 20% to 80%. Different speed-ups among the selectivity range in Figure 5.12 originate from different run-time trends shown in Figure 5.1. Run-time trends change with increasing parallelism from a curve with a peak and falling edges to a curve with a steep increase followed by a constant pathway. Besides different speed-up curves, selections scale non-linearly for larger DOPs. Using 8 and 12 physical cores, the linear speed-up is only reached for medium selectivities and this range becomes smaller with increasing DOP. Finally, if hyper-threading is applied for 16, 20, and 24 logical cores, the speed-up becomes sub-linear for the entire selectivity range.

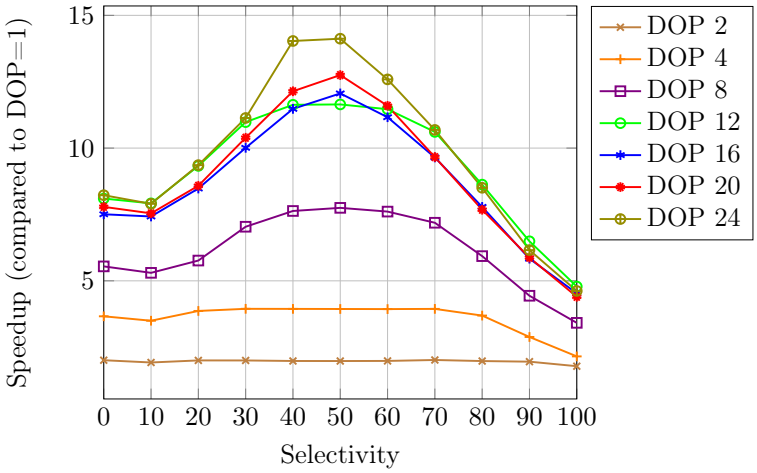


Figure 5.12: Speedup.

5.8 Cost Models

There are two common approaches to determine costs for a selection. Ross [Ros04] determines costs based on induced branch mispredictions. This approach claims, that branch mispredictions are the main contributor to the run-time of a selection. In contrast, Pirk et al. [Pir13] determine costs based

on induced cache accesses because they claim that cache accesses are the main contributor.

Both cost models consist of two parts. First, they predict the number of performance impacting events according to their assumption about the main performance contributor. Then, they multiply these numbers by a penalty which estimates the cost per event. Whereas both models predict the occurrence of their performance impacting event very precisely, resulting run-time estimations are more inaccurate. In Figure 5.13, we show estimated as well as measured cycles for different DOPs. Although both cost models do not take parallelism into account, their estimation could not be adjusted by a scalar factor to estimate the entire range of parallel execution. Ross [Ros04] estimates cycles as well as trends more precisely for low DOPs because they take branch misprediction into account. As our evaluation shows, branch mispredictions are the major cost contributor for a selection using one core (see Figure 5.7.2). In contrast, Pirk et al. [Pir13] estimate a different trend which correlates to parallel execution using high DOPs because they take L3 accesses into account. As our evaluation shows, L3 accesses are the major cost contributor for a selection using all logical cores (see Figure 5.7.2). Note, neither of these cost models take L3 misses into account and thus they misestimates a selection using all physical cores (12).

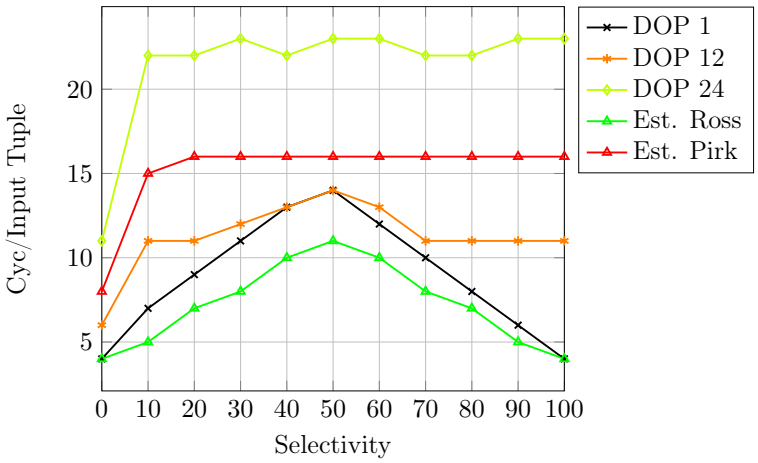


Figure 5.13: Selection Costs.

In summary, both cost models take only one performance contributor into account and therefore fail to predict run-times correctly as well as their trends for different DOPs. Overall, our evaluation reveals, that characteristics of modern CPUs are too complex and interrelated such that a cost model based on one performance contributor is not able to model the entire range of parallelization. However, the cost model by Ross [Ros04] and Pirk et al. [Pir13] can be used as reference points for low and high DOPs, respectively. We argue, that modern CPUs require a combined cost model

which takes multiple characteristics into account to reflect the interrelated processor characteristics of modern CPUs.

5.9 Summary

In this chapter, we extensively studied performance characteristic of selections on modern CPUs. We showed, that branch mispredictions as well as cache accesses can be predicted by common cost models. However, these models fall short to estimate run-times for different DOPs. By revealing deterministic behaviors of modern CPUs, we pave the way for more accurate cost estimations in DBMS. Furthermore, our insights from an in-depth selection analysis can be used to improve cost estimations for other DBMS operators such as projections, joins, or aggregations.

Based on our results, future work could create a cost model that takes branch mispredictions as well as cache accesses with different weighting into account. The weighting will depend on the number of executing cores and the cost per event in individual CPU components. Furthermore, such an in-depth analysis for other DBMS operators could improve the cost estimation of database optimizers significantly.

Chapter 6

Counter-Based Query Execution

Progressive optimization introduces robustness for database workloads towards wrong estimates, skewed data, correlated attributes, or outdated statistics. Previous work focuses on cardinality estimates and rely on expensive counting methods as well as complex learning algorithms.

In this chapter, we utilize performance counters to drive progressive optimization during query execution. The main advantages are that performance counters introduce virtually no costs on modern CPUs and their usage enables a non-invasive monitoring. We present fine-grained cost models to detect differences between estimates and actual costs which enables us to kick-start re-optimization. Based on our cost models, we implement an optimization approach that estimates the individual selectivities of a multi-selection query efficiently. Furthermore, we showcase that foreign key joins and expensive predicate evaluations can also be optimized using our approach. Finally, we are able to learn properties like sortedness, skew, or correlation during run-time.

In our evaluation we show, that the overhead of our approach is negligible while the performance improvements are convincing. Using progressive optimization, we improve run-time up to a factor of three compared to average run-time and up to a factor of 4,5 compared to worst case run-time. As a result, we avoid costly operator execution orders and thus make query execution highly robust.

The rest of this chapter is structured as follows: At first, we introduce progressive optimization for DBMS in Section 6.1 and show related work in Section 6.2. Then, we provide necessary background on efficient in-memory data processing in Section 6.3. In Section 6.4, we present the underlying hardware-conscious cost models for our optimization approach. We describe our optimization approach in Section 6.5 and evaluate our approach in Section 6.6. Finally, we conclude in Section 6.7.

6.1 Progressive Optimization for Databases

The migration of databases from disk to faster memories such as RAM, Flash, or NVRAM fundamentally changes the cost balance in analytical data management systems: where disk-based system performance was largely dominated by disk bandwidth and latency, in-memory analytics systems have to consider CPU efficiency as a major contributor to performance. This requires a careful (re-)investigation of various design decisions underlying *classic* relational DBMSs with respect to the new considerations. This re-investigation has been done for many components of the classic analytical database design such as processing [Bea05, MBK02], indexing [LKN13, LLS11], and compression [Rea13]. However, most of these techniques were developed to address hardware-specific cost factors such as cache thrashing, misprediction penalties, and synchronization costs. Therefore, many of these techniques use new hardware features such as SIMD instructions, transactional memory, or deep memory hierarchies in order to overcome hardware challenges.

In this chapter, we re-investigate the idea of progressive optimization [Mea04] on modern processors. With progressive optimization, a physical query plan is adapted to the characteristics of the currently processed data subset. Following previous work, our approach is based on monitoring and analysis. However, unlike previous work, our approach has virtually no CPU costs by making extensive use of a handy feature of modern CPUs: the *Performance Monitoring Unit* [Int12b]. This unit allows the counting of performance-related events such as retired instructions, cache-misses, and branch mispredictions. By comparing the sampled performance counters with expected numbers of fine-grained cost models at run-time, we might detect differences of the estimated from the actual costs; thus, possibly kick-starting a re-optimization process. In fact, our approach effectively renders high quality decisions at query compilation time unnecessary because it provides better and more adaptive information at run-time. In addition to low CPU-overhead, such non-invasive monitoring extends the applicability of progressive optimization to cases when instrumentation is not an option such as binary UDFs or calls to external libraries. These benefits, however, hinge on the availability of appropriately accurate cost models. Consequently, our specific contributions include:

- an unified cost model for memory accesses as well as branch misprediction costs in modern CPUs,
- an estimation component that derives data-specific characteristics such as selectivities and domains from performance event counters using non-linear optimization,
- a run-time execution component which balances the trade-off between the quality of the estimation and the required optimization time.

To illustrate the importance of avoiding bad plans when evaluating analytical queries on memory-resident data, we compare the cost of the worst and the best physical plan for Query 6 of the TPC-H benchmark:

```
SELECT sum(l_extendedprice * l_discount) as revenue
FROM lineitem
WHERE l_shipdate <= VALUE and l_quantity < 24
and l_discount between 0.06 - 0.01 and 0.06 + 0.01
```

We implemented Query 6 in C and use the order of the four selection predicates and the selectivity of the ship date predicate as degrees of freedom. In Figure 6.1, we plot the run-time difference between the fastest and slowest plan on the y-axis. As shown, it is important to select an appropriate plan, especially when the selectivity of the ship date condition is low. Furthermore, real life databases are bulk loaded and, hence, weakly clustered on the date column. As a consequence, the selectivity varies over the course of the table and thus different plans with different predicate orders are optimal for different phases of the scan. Thus, quickly recognizing when a good plan has gone bad requires fine-grained monitoring. In the remainder of this chapter, we describe an approach to perform such monitoring at negligible overhead.

6.2 Related Work on Progressive Optimization

Previous work on progressive optimization by Markl et al. [Mea04], Kabra et al. [KD98], and Babu et al. [BBD05] validates cardinality estimates against actual values measured during run-time execution. If a significant disagree-

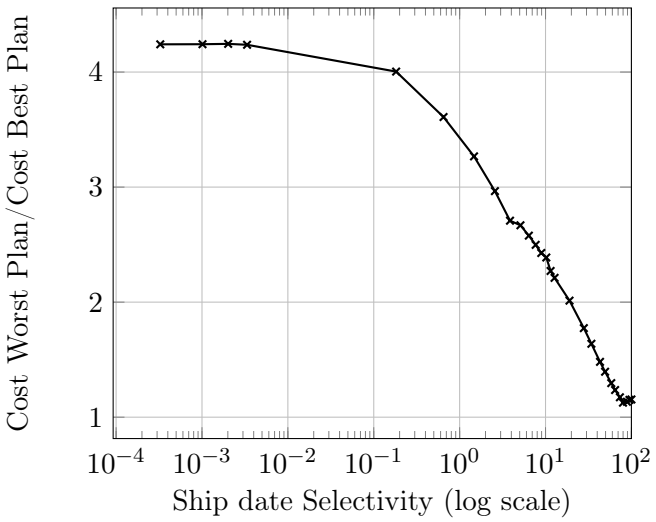


Figure 6.1: Best vs. Worst Plan costs for TPC-H Query 6.

ment is detected, the query execution might stop and a re-optimization process is triggered. Kache et al. [Kea06] extends this approach for federated databases and Han et al. [Hea07a] for shared-nothing parallel databases. These approaches can be collectively termed as plan-switching approaches, as they involve run-time switching among complete query plans. In contrast to these approaches, we base our re-optimization decision on actual performance counters which induce virtually no costs. Furthermore, we progressively optimize the current execution by inspecting the query vector-wise. This enables us to perform a more fine-grained optimization. Finally, we are able to exploit more properties than just the cardinality to re-optimize query plans and do not require any statistics over the data.

With the LEarning Optimizer LEO, Stillger et al. [Sea01] presents an approach to repair incorrect statistic and cardinality estimates. By monitoring previously executed queries, LEO computes adjustments based on the difference between optimizer estimates and actual measured costs. In contrast, our approach learns from the vector-wise processing of the same query to optimize future vector executions. Thus, we provide a feedback loop during run-time as opposed to LEOs feedback loop among multiple query executions.

Răducanu et al. [RBZ13] propose a micro adaptivity approach to learn the best implementation of a function during run-time. Therefore, they measure the run-time of different function implementations and apply a learning algorithm to choose the most promising implementation. In contrast, by sampling performance counters instead of run-time or even incremental tuple counters, we are able to learn properties of the data sets like sortedness or co-clusteredness of joins. Furthermore, we significantly reduce the overhead during run-time and provide a non-invasive approach. Finally, Răducanu et al. [RBZ13] choose an alternative implementation randomly from a pool of functions. In contrast, we infer selectivity of individual attributes and thus converge to the optimal plan faster.

Another research area discovers the best QEP based on a subset of possible best plans. Dutt et al. [DH14] propose to exploit a *bouquet* of plans from a set of optimal plans such that at least one of this plans is near-optimal. In contrast to our approach, they require more overhead during compile-time as well as during run-time. During compile time, they have to gather the bouquet of plans. In contrast, we create different orderings of operators during run-time using JIT compilation. During run-time, Dutt et al. [DH14] introduce explicit counters between operators. In contrast, we exploit performance counter which nearly induce no costs.

6.3 Background

This section provides the necessary background for measuring and estimating query execution performance. In Section 6.3.1, we start by describing how to transform a query expressed by relational algebra into code that can be executed by a machine. Then, Section 6.3.2 introduces branch-related and cache-related counters that enable us to reason about hardware utilization of modern CPUs.

6.3.1 From Relational Algebra to Machine Code

In this section, we describe how a DBMS transforms a query into executable code using *Just-In-Time* compilation like Hyper [Neu11]. For this transformation, we use the following query on the TPC-H data set. The query calculates the sum of discounts for all lineitems with a quantity less than 100 and a ship date before February 2, 1992.

```
Select sum(discount) from lineitem
where quantitiy <=100
and shipdate <= '1992-02-02'
```

This query can be transformed into the following code written in C (assuming a column-oriented data layout). We emphasize that, we convert the ship date column from date to time-stamp to replace an expensive string comparison with a cheaper integer comparison.

```
for (int i = 0; i < lineitem.size(); i++)
   if(quantity[i] <= 100)
      if(shipdate[i] <= timeStamp)
         sum += discount[i];
```

This C-program iterates over all elements in the lineitem table. For each $tuple_i$, it first checks if its quantity attribute is less or equal to 100. If $tuple_i$ qualifies, its second attribute is evaluated by the second predicate. If the ship date of $tuple_i$ is before or equal to 1992-02-02 and thus the second predicate qualifies, its discount is added to the overall sum. In general, each predicate evaluation introduces a branch with two possible outcomes. From a performance perspective, a selection induces three important characteristics. First, the *quantity* attribute as the first predicate is accessed for each tuple, regardless of its selectivity. Second, the number of accesses to the second attribute *ship date* depends on the selectivity of the first predicate. Therefore, ship date is only evaluated for qualifying tuples of the previous predicate on the quantity attribute. Third, the access to the third column *discount* depends on the selectivity of the second predicate and is only evaluated if all preceding predicates qualify that tuple.

For this transformation, we choose one possible QEP that evaluates the quantity predicate first. However, we could also evaluate the ship date predicate first to create another QEP. In the remainder of this chapter, we refer

to each possible order of a multi-selection query as one *predicate evaluation order* (PEO).

In a final step, a compiler translates the C-code into machine instructions. For each predicate evaluation, the compiler generates one comparison followed by a conditional jump instruction. Additionally, one such pair and an increment instruction for the loop counter is generated for the entire loop. The conditional jump determines the following execution path. If a tuple qualifies, the branch/jump is *not taken* and thus the execution continues with the next instruction. In contrast, if a tuple does not qualify, a branch is *taken* and therefore the program execution jumps to the end of the loop code to test the loop condition. In the latter case, the subsequent instructions to check the second predicate and update the sum are omitted. We refer to Section 5.2 for a detailed description of this transformation step.

6.3.2 Performance Counters

In this section, we introduce branch-related (Section 6.3.2.1) and cache-related (see Section 6.3.2.2) performance counters which allow us to reason about the performance of a QEP. Modern CPUs provide dynamic data obtained from so-called *performance monitoring units (PMU)* to measure the CPU and system resource utilization. Performance counters can be divided into *constant* counters that do not change their values among all possible PEOs and *mutable* counters. The number of branches taken is constant among all PEOs because all PEOs of the same QEP lead to the same query result and thus induce the same number of qualifying tuples. In contrast, the number of conditional branches, branches not taken, and cache-related counters, are mutable and thus vary among PEOs.

6.3.2.1 Branch-related Counters

Branches strongly impact the query performance on modern CPUs. Therefore, CPUs possess a dedicated *branch prediction unit* [Int12b] which tries to predict the outcome of each branch. A wrongly predicted branch leads to pipeline flushes, poor instruction cache locality, and limited instruction level parallelism [Aea99]. Ross [Ros04] investigated this effect for multi-selection queries and show, that the branch predictor correctly predicts branches for queries with very high or very low selectivities. On the other hand, queries with medium selectivities lead to many incorrect predictions which accumulate to the worst-case prediction behavior for a selectivity of 50%. The branch-related performance counters in modern CPUs allow us to capture this behavior by counting the number of right and wrong branch predictions. Furthermore, we may divide mispredictions into branches that are mispredicted as taken and branches that are mispredicted as not taken. Finally, PMUs are able to count the number of branches taken and not taken as well

as their sum as the number of conditional branches [Int12b]. In Figure 6.2, we plot these counters for a single selection query with varying selectivity. Whereas branch misprediction counters depend on CPU internal branching algorithms, branch taken/not taken counters depend solely on the generated code and thus they are independent of CPU characteristics such as prefetching or *out-of-order* execution.

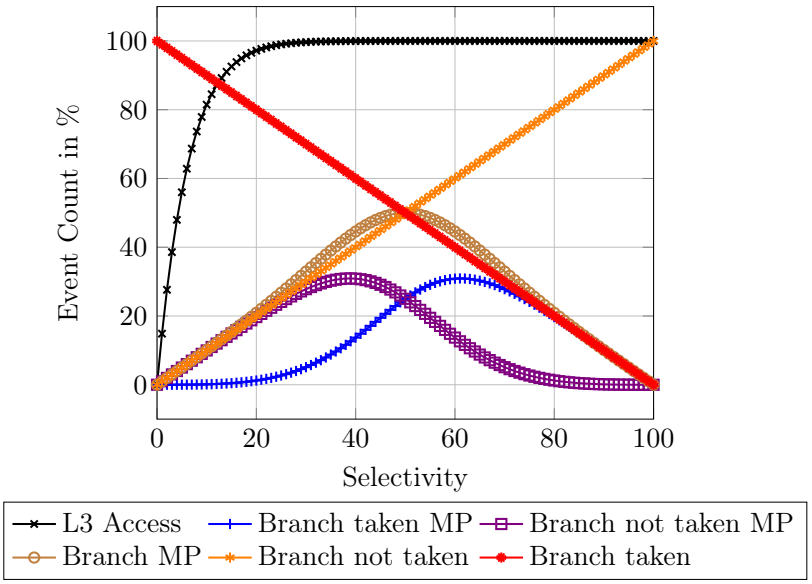


Figure 6.2: Performance Counter Overview.

We exploit the number of branches taken b_T to determine the number of qualified tuples by a PEO. If all predicates qualify, only one branch is taken at the end of the loop. In contrast, if one predicate does not qualify, two branches are taken (one to test the loop condition and one back to the beginning of the loop). Using n as the number of tuples, we calculate the number of qualifying tuples by $2 * n - b_T$.

We exploit the number of branches not taken b_{NT} to determine characteristics of individual predicates during run-time. Each tuple induces between zero and p not taken branches with p as the number of predicates. Zero not taken branches are induced if the first predicate does not qualify. In contrast, p not taken branches are induced if all predicates qualify. In between these boundaries, each descend of a tuple in the PEO increments the branch not taken counter by one for every predicate that qualifies. As a general performance rule, the less branches are not taken by a PEO, the better the performance will be. The main reason is that each predicate evaluation induces additional work in terms of computation, memory accesses, and branching costs. We refer to Section 5.4 for a detailed discussion on branching counters for selections.

6.3.2.2 Cache-related Counters

The memory hierarchy of modern CPUs consists of registers, multiple layers of caches, main memory, and disks. In our approach, we focus on the utilization of the multi-level cache hierarchy to improve the performance of modern in-memory databases.

For our approach, we exploit the number of accesses to the L3 cache because they add up demand requests from upper cache levels as well as prefetching requests from the L1 or L2 prefetcher units. In contrast to L3 hits and misses, the number of L3 accesses are independent of CPU characteristics such as prefetching algorithms or *out-of-order* execution. In this chapter, we focus on multi-selection queries that exhibit no tuple reuse in general. Therefore, the number of L3 accesses are equal to the number of accesses to L1 and L2 plus prefetch accesses.

For a multi-selection query, the demand on the memory bus depends on the number of predicate evaluations. In general, a subsequent load and compare must be executed if the current predicate qualifies the tuple at hand. Thus, the selectivity as well as the order of the individual predicate evaluations impact the number of load operations. However, selectivities cannot be changed because they are determined by the individual predicates and the value distribution of the data set. Therefore, the PEO remains the most important query optimization parameter.

Finally, the utilization of the memory hierarchy in modern CPUs is impacted by speculative execution and prefetching. Speculative execution predicts the outcome of a branch, i. e., if a tuple qualifies or not. If the prediction is correct, speculatively loaded instructions and data are executed earlier in time and thus the execution is accelerated. However, a wrong prediction induces unnecessary memory accesses and executes expendable instructions. Prefetching on the other hand tries to recognize memory access patterns and prefetches expected memory accesses [Int12b]. Similar to speculative execution, a wrong prefetch induces unnecessary memory accesses and a correct prediction accelerates execution. We refer to Section 5.5 for a detailed discussion on cache counters for selections.

6.4 Cost Models

In this section, we present the underlying cost models of our approach. We introduce a model for cache accesses in Section 6.4.1 and a model for branch mispredictions in Section 6.4.2. Using our cost formulas, we are able to model all performance counters shown in Figure 6.2.

6.4.1 Cache Cost Model

The generic cost model by Manegold et al. [Man02] allows us to predict the number of cache misses for different access patterns. Furthermore, by com-

bining access patterns, complex relational operators such as joins or sorts can be modeled. For our approach, we utilize the extension of the generic cost model by Pirk et al. [Pir13] to model the cache accesses of different PEOs. We estimate induced cache accesses of a multi-selection query by exploiting two patterns. The first predicate introduces a *single sequential access pattern* which induces one random access for accessing the first cache line and one sequential access for each subsequent cache line. Each subsequent predicate introduces a *sequential scan with conditional read pattern* which induces cache accesses depending on the selectivity of the previous predicate.

Using the cost formula by Pirk et al. [Pir13], we predict the number of L3 accesses. As shown in Figure 6.2, the number of L3 accesses sharply increase for small selectivities below 20% and then remain constant. The main reason for this behavior is the high number of random misses for small selectivities. These random misses occur because cache lines are omitted. In contrast, with increasing selectivity, the access probability per cache line increases and thus less cache lines are omitted. This behavior is reflected by the reduced number of cache line accesses that are only present in the range of 0-20% selectivity. For a selectivity larger than 20%, each cache line is accessed and thus the number of cache line accesses remains constant. This characteristic also applies for a multi-selection query.

Following Pirk et al. [Pir13], we determine the number of L3 accesses with B_i as the cache line size in words and $|R_{B_i}|$ as the number of cache lines covered by a column:

$$P_i = 1 - (1 - selectivity)^{B_i} \tag{6.1a}$$

$$P_i^s = (1 - (1 - selectivity)^{B_i})^2 \tag{6.1b}$$

$$P_i^r = (1 - s)^{B_i} - (1 - s)^{2B_i} \tag{6.1c}$$

$$M_i^s(s\_trav\_cr) = P_i^s * |R_{B_i}| \tag{6.1d}$$

$$M_i^r(s\_trav\_cr) = (P_i - P_i^s) * |R_{B_i}| \tag{6.1e}$$

With P_i, we refer to the probability of accessing a cache line when traversing it. It is equal to the probability that any of the data items of the cache line is accessed. Furthermore, P_i can be distinguished into the probability that a cache miss is a sequential miss P_i^s or a random miss P_i^r. By exploiting the probability of a cache line access, we can estimate the number of cache misses for an access pattern in terms of sequential access misses M_i^s and random access misses M_i^r.

Based on an extended evaluation of the cost model on modern CPUs, we modify the cost model by Pirk et al. [Pir13] to double count the number of random misses which leads to more precise estimations. This modification models the effect, that a random cache miss induces one cache access for the cache line that was predicted but not used and one cache line access for the actually used cache line.

Finally, we provide a cost formula to model the cache accesses for equi-joins. Equi-Joins tend to be dominated by memory access costs. Therefore, we use memory access costs as the primary metric for distinguishing between different join algorithms. There are two main factors that determine the relative costs of a sequence of join operators: the number of accesses and their locality. The number of accesses is determined by the selectivity of the operators preceding a join while the locality is a property of the underlying data distribution which is determined when loading the data. To effectively optimize the order of joins for our cost-based approach we have to take these factors into account. For that purpose, the generic cost model by Manegold et al. [Man02] contains equations to predict the number and type (random or sequential) of cache misses. However, our evaluation shows, that the equation for the number of cache misses in the original cost model was highly inaccurate. Therefore, we develop an alternative equation that yields significantly better predictions and is grounded in the external memory model [AV88]. In Equation 6.2, we combine the original model for sequential cache misses and a multiplicative factor to model random cache misses.

$$M_i^r = \begin{cases} C_i & \text{if } C_i < \#_i \\ r * \left(1 - \frac{\#_i \cdot B_i}{R.n \cdot R.w}\right) & \text{if } C_i \geq \#_i \end{cases} \tag{6.2}$$

The number of accessed cache lines (C_i) is calculated from the size of the relation ($R.n$), the width of a tuple in bytes ($R.w$), the number of accesses (r), the cache parameters line size (B_i), and cache capacity in lines ($\#_i$):

$$C_i = R.n \cdot \left(1 - \left(1 - \frac{1}{R.n}\right)^r\right) \tag{6.3}$$

6.4.2 Branch Cost Model

In Section 5.4 we pointed out, that branch mispredictions follow the number of branches not taken for a selectivity below 50% and the number of branches taken for a selectivity above 50%. Thus, for a selection with a selectivity below 50%, the branch predictor predicts that each tuple does not qualify (branch is taken) and therefore mispredicts each qualifying tuple (branch not taken). Hence, the number of branch mispredictions is equal to the number of branches not taken. On the other hand, for a selection with a selectivity above 50%, the branch predictor predicts that each tuple qualifies (branch is not taken) and thus mispredicts each not qualifying tuple (branch taken). Based on this observation, we calculated the number of branch mispredictions (BRMP) using the number of branches not taken (BNT) in Section 5.4 by:

$$BRMP(p) = \begin{cases} BNT(p), \text{if p} \leq 0.5 \\ BNT(1-p), \text{if p} > 0.5 \end{cases} \tag{6.4}$$

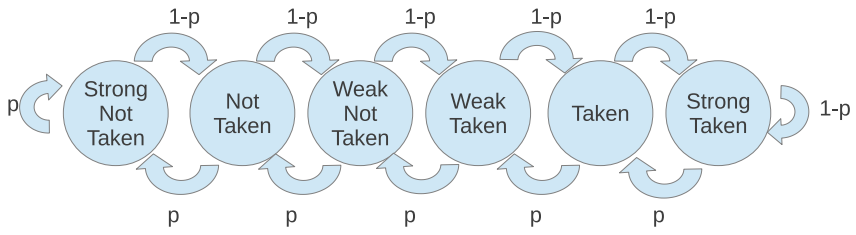


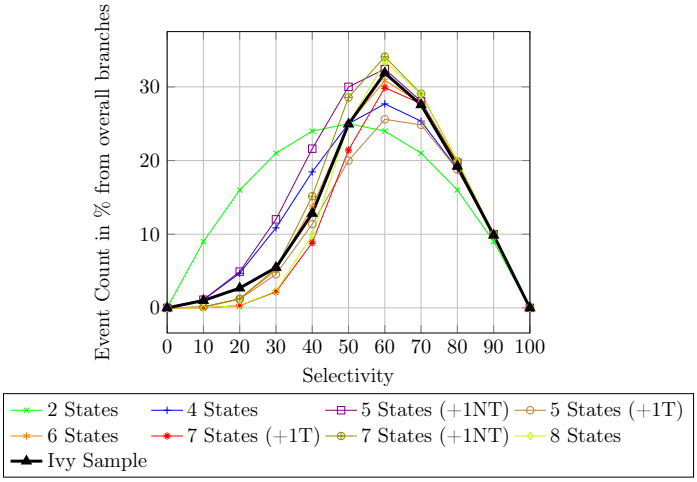
Figure 6.3: Markov Chain.

As shown in Figure 5.3, this estimation becomes inaccurate in the selectivity range around 50%. Therefore, in this chapter, we propose a Markov chain to model the branching behavior of modern CPUs. This Markov chain uses a stationary distribution given the selectivity p as the transition probability. In Figure 6.3, we show a six-state Markov chain. In the first three states, the branch predictor predicts the branch to be *not taken*. In contrast, in the last three states, the branch predictor predicts the branch to be *taken*.

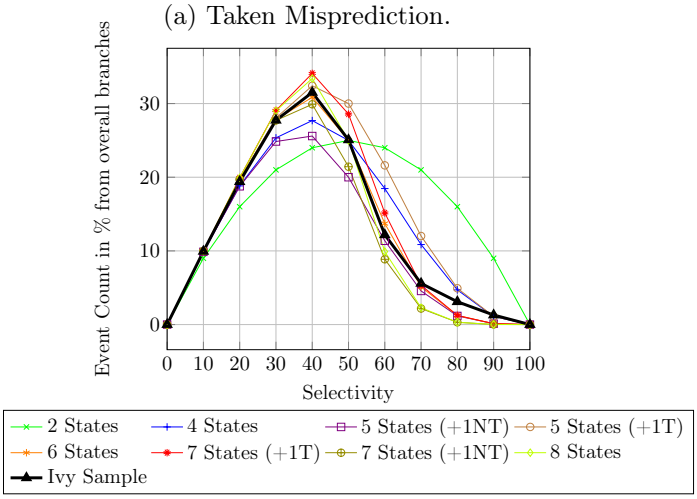
The probability of a transition from one state to another is determined by the selectivity p. With a probability of p, a branch is not taken and the current state will transits one state to the left. In contrast, with a probability of $p-1$, a branch is taken and the current state will transits one state to the right. A Markov chain allows us to predict the number of mispredictions as well as distinguish them into branches that are mispredicted as taken and branches that are mispredicted as not taken.

In Figure 6.4, we compare Markov chains using a different number of states ranging from two to eight. Additionally, we introduce an uneven state count which favors either branches taken (+1T) or branches not taken (+1NT) by adding an additional state. We compare these predictions against real occurrences on a Ivy-Bridge CPU. Our evaluation in Section 5.4 showed, that branching algorithms for the execution of a selection were not changed over the last three micro-architectures Sandy-Bridge, Ivy-Bridge, and Haswell. Thus, we show only real occurrences on the Ivy-Bridge micro-architecture.

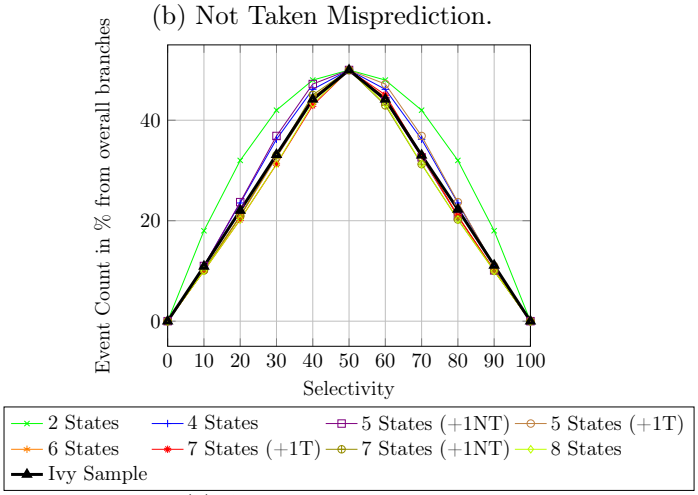
As shown in Figure 6.4, the six state Markov chain estimates the number of taken and not taken branches as well as their sum almost exactly. Therefore, we use a six state Markov chain in the remainder of this chapter. Considering the number of all branch mispredictions (see Figure 6.4c), other state counts produce good predictions too. However, these state counts underestimates or overestimates branches taken/not taken. Thus, they underestimate one event in the same portion as they overestimate the other. Interestingly, the peak occurrence of mispredicted taken/not taken branches are shifted by 10% percent (to 40% or 60% selectivity) as opposed to the overall number of mispredictions. Finally, on AMD CPUs, we observe the most precise prediction using four states.



(a) Taken Misprediction.



(b) Not Taken Misprediction.



(c) All Misprediction.

Figure 6.4: Markov Chain Bits.

To calculate the branch taken/not taken mispredictions, we solve the following system of equations. Intuitively, the probability that a current state is reached is composed of the probability of the previous and subsequent state multiplied by the probability that these states change to the current state. We label the states as *strong not taken (SNT)*, *not taken (NT)*, *weak not taken (WNT)*, *weak taken (WT)*, *taken (T)*, and *strong taken (ST)*.

$$SNT = SNT * p + NT * p \tag{6.5a}$$
$$NT = SNT * (1-p) + WNT * p \tag{6.5b}$$
$$WNT = NT * (1-p) + WT * p \tag{6.5c}$$
$$WT = WNT * (1-p) + T * p \tag{6.5d}$$
$$T = WT * (1-p) + ST * p \tag{6.5e}$$
$$ST = ST * (1-p) + T * (1-p) * p \tag{6.5f}$$
$$SNT + NT + WNT + WT + T + ST = 1 \tag{6.5g}$$

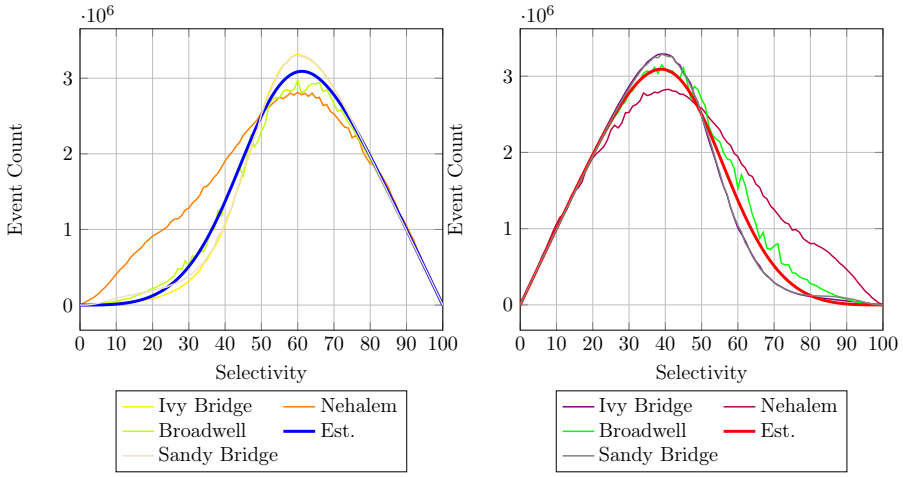
By solving this system of linear equations, we receive the formulas to calculate the probability that a selection with a selectivity p is in a certain state.

$$SNT = \frac{p^5}{3*p^4 - 6*p^3 + 7*p^2 - 4*p + 1} \tag{6.6a}$$
$$NT = \frac{-p^5 + p^4}{3*p^4 - 6*p^3 + 7*p^2 - 4*p + 1} \tag{6.6b}$$
$$WNT = \frac{p^5 - 2*p^4 + p^3}{3*p^4 - 6*p^3 + 7*p^2 - 4*p + 1} \tag{6.6c}$$
$$WT = \frac{-p^5 + 3*p^4 - 3*p^3 + p^2}{3*p^4 - 6*p^3 + 7*p^2 - 4*p + 1} \tag{6.6d}$$
$$T = \frac{p*(p-1)^4}{3*p^4 - 6*p^3 + 7*p^2 - 4*p + 1} \tag{6.6e}$$
$$ST = \frac{-(p-1)^5}{3*p^4 - 6*p^3 + 7*p^2 - 4*p + 1} \tag{6.6f}$$

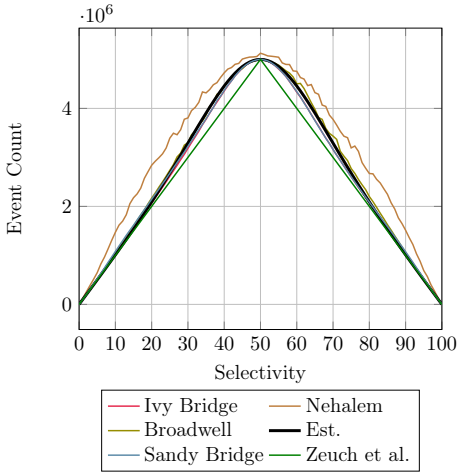
Using the probabilities of individual states, we sum up the probability that a branch is taken by $B_{Tak} = WT + T + ST$ and the probability that a branch is not taken by $B_{NotTak} = SNT + NT + WNT$. Based on these probabilities, we determine the following estimation formulas for mispredictions (MP) and right predictions (RP). We calculate mispredicted branches taken B_{TakMP} and right predicted branches taken B_{TakMP}, mispredicted branches not taken $B_{NotTakMP}$ and right predicted branches not taken $B_{NotTakRP}$, and the sum of all mispredictions B_{MP} and right predictions B_{RP}. Note that we determine the actual number of mispredictions by multiplying these probabilities with the number of input tuples.

$$B_{TakMP} = (1 - p) * B_{NotTak} \tag{6.7a}$$
$$B_{TakRP} = (1 - p) * B_{Tak} \tag{6.7b}$$
$$B_{NotTakMP} = p * B_{Tak} \tag{6.7c}$$
$$B_{NotTakRP} = p * B_{NotTak} \tag{6.7d}$$
$$B_{MP} = B_{TakMP} + B_{NotTakRP} \tag{6.7e}$$
$$B_{RP} = B_{TakRP} + B_{NotTakRP} \tag{6.7f}$$

In Figure 6.5, we evaluate our prediction formulas against the latest Intel micro-architectures Nehalem, Sandy-Bridge, Ivy-Bridge, and Broadwell for a selection on 10M tuples.



(a) Branch Taken Misprediction. (b) Branch Not Taken Misprediction.



(c) Branch Misprediction Overall.

Figure 6.5: Estimated vs. Measured Branch Counter Overview.

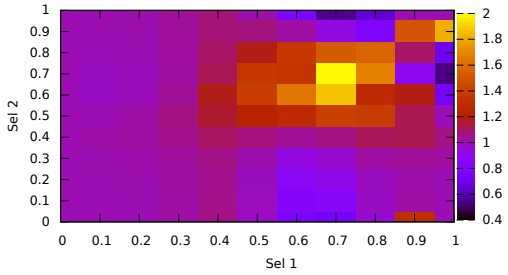
As shown, only Nehalem as the oldest micro-architecture partially differs from our predictions. In contrast, our prediction fits real occurrences on Sandy-Bridge, Ivy-Bridge, and Broadwell quite well. In particular, the overall branch mispredictions are estimated very precisely. However, in the selectivity range around 40% and 60%, there are minor deviations but the overall trend is predicted correctly. Compared to our simple estimation in Section 5.4, we present a more accurate estimation that is able to distinguish between mispredicted taken and not taken branches.

For a multi-selection query, we extend our branch estimations to model each predicate $p_1...p_n$. Therefore, we replace the number of input tuples by the number of output tuples of the previous predicate. In Figure 6.6, we present branch estimations for a selection using two predicates as a 2D heat map. Each axis plots the selectivity of one predicate. At the interception point of two selectivities, we plot the relationship between the measured performance counter and our estimation. As shown, mispredicted branches not taken are underestimated slightly in the selectivity range of 60-80% for both predicates (see Figure 6.6a). In contrast, mispredicted branches taken are slightly overestimated in the selectivity range of 20-40% for the first predicate (see Figure 6.6b). Finally, overall branch mispredictions have a minor underestimation in the range of 60-80% for both predicates but the overall estimation differs in less than 10% (see Figure 6.6c). Despite some outliers, we predict branch events for multi-selection queries very precisely with only minor differences in some selectivity ranges.

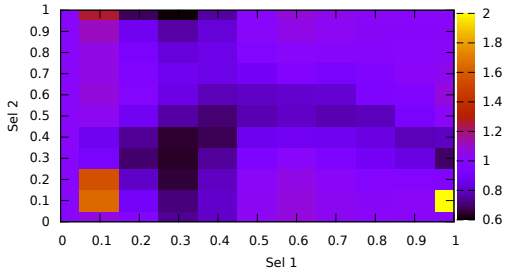
6.5 Optimization Approach

In this section, we present our progressive optimization approach which exploits the cost models presented in the previous section (see Section 6.4). Progressive optimization is valuable because determining the *best* predicate evaluation order (PEO) at compile time is rarely possible. The main reasons are uncertain or imprecise information at compile-time such as wrong cardinality estimates, skewed data, correlated attributes, outdated statistics, or user-defined functions which may lead to sub-optimal decisions [KD98]. Our optimization algorithm alleviates these uncertainties by deriving the selectivity of individual predicates during run-time.

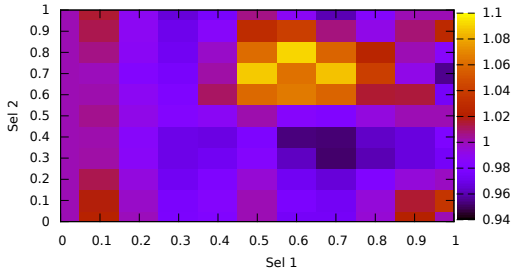
We present our progressive optimization approach in three parts. First, we introduce the search space restriction in Section 6.5.1 that allows us to prune some areas of the search space. Second, we introduce the non-linear optimization algorithm that explores the pruned search space and our cost models to estimate individual predicate selectivities (Section 6.5.2). Third, we introduce an algorithm to create different start points inside the pruned search space for the optimization algorithm (Section 6.5.3). Finally, we summarize the entire optimization process in Section 6.5.4 before examining the impact of skew and correlation on our approach in Section 6.5.5.



(a) Measured/Predicted Not Taken Branch Mispredictions.



(b) Measured/Predicted Taken Branch Mispredictions.



(c) Measured/Predicted Branch Mispredictions.

Figure 6.6: Two Predicate Branch Mispredictions.

6.5.1 Search Space Restriction

The initial search space for a given query with p predicates encompasses a $p-$dimensional space with a possible selectivity between zero an 100% for each predicate. By exploiting the number of input and output tuples of a query, we might restrict this search space. The searched query has to be between the *upper* and *lower* tuple bound. Intuitively, the upper

tuple bound represents the highest number of accesses to $col_1 ... col_n$ that is possible considering the given number of input tuples $tups_{in}$ and output tuples $tups_{out}$. In contrast, the lower tuple bound represents the lowest number of accesses to these columns. We define the number of accesses to $col_1 ... col_n$ by predicate $p_1 ... p_n$ for the upper and lower tuple bound as:

$$Upper\,Tuple\,Bound(p) = \begin{cases} tups_{out}, \text{if p = n} \\ tups_{in}, \text{else} \end{cases} \tag{6.8}$$

$$Lower\,Tuple\,Bound(p) = tups_{out} \tag{6.9}$$

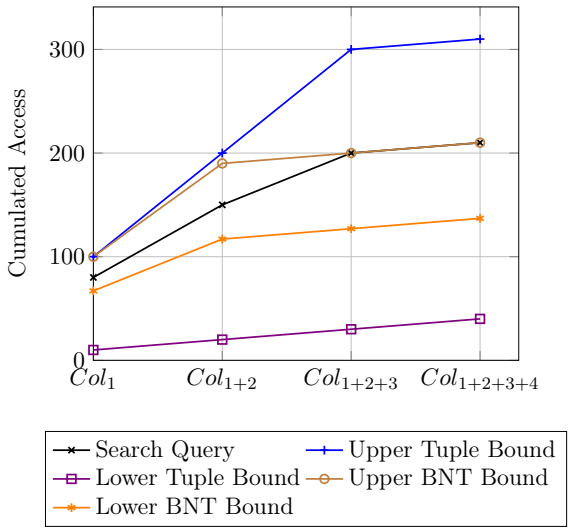


Figure 6.7: Search Space Restriction.

In Figure 6.7, we restrict the search space of an example query. The search query consists of four predicates that select 10 output tuples from 100 input tuples. The accesses to $[col_1, ..., col_4]$ are $[80, 70, 50, 10]$. The sum of these accesses (210) is equal to the number of branches not taken. Using the upper and lower tuple bound, we restrict the possible access intervals for $[col_1, ..., col_4]$ to the lower bound $[10, 10, 10, 10]$ and the upper bound $[100, 100, 100, 10]$. Note, Figure 6.7 shows the cumulative accesses for our example.

To restrict the search space further, we exploit the number of branches not taken (BNT). The number of branches not taken are independent of run-time or CPU characteristics. Thus, the same number of branches not taken originate on each CPU. Furthermore, branches not taken exhibit two important characteristics. First, branches not taken by predicate p_i represent the number of accesses to column col_i. Second, we can sample the number of branches not taken for an entire PEO and they correspond to the sum of the

accesses to $col_1...col_n$. Therefore, the sampled number of branches not taken represents a definite integral among accesses to $col_1...col_n$. Additionally, accesses to the first and last column exhibit special characteristics. All tuples in the first column col_0 are always accessed; thus, we define $access(col_0) = tups_{in}$. In contrast, tuples in the last column col_n are only accessed for each output tuple; thus, we define $access(col_n) = tups_{out}$. Note, in the following, we argue among accesses to individual columns which can be converted into the selectivity product of $p_1...p_i$ by $\prod_{p=1}^{i} = \frac{Acc(col_i)}{tups_{in}}$. Using the selectivity product, we determine individual predicate selectivity by $p_i = \frac{\prod_{p=1}^{i} p}{\prod_{p=1}^{i-1} p}$.

We exploit the aforementioned characteristics to further restrict the search space of a search query. At first, cumulative accesses to $col_1...col_n$ match the sampled number of branches not taken. Thus, a query that introduces either more or less branches not taken cannot be the search query. Based on the sampled branches not taken and the special characteristics that accesses to col_i can only be less or equal to accesses to col_{i-1}, we can restrict the search space by a new lower and upper bound on the number accesses per column.

An *upper BNT bound* is defined by assigning accesses to $p_1..p_n$ such that p_i can access the maximum number of tuples. The maximum number of accesses by p_i requires that all previous predicates $p_0...p_{i-1}$ access as many tuples as p_i. Otherwise, the constraint that p_i is only allowed to access less or equal tuples as p_{i-1} would be violated. The remaining predicates $p_{i+1}...p_n$ access the minimum number of tuples ($tups_{out}$). If the maximum number of accesses by p_i exceeds the number of input tuples, we restrict p_i to $tups_{in}$ because no predicate can access more tuples than exist. In Figure 6.7, each query that has one sample point above the upper BNT bound cannot reach the desired number of output tuples. This would require a predicate to access less tuples than the number of output tuples. Thus, we define the upper BNT bound using BNT_{samp} as the sampled branches not taken:

$$Upper\,BNT\,Bound(p) = \begin{cases} tups_{out}, \text{if p = n} \\ tups_{in}, \text{if Upper BNT Bound(p)} > tups_{in} \\ \frac{BNT_{samp} - (tups_{out} * (n-p-1))}{p+1}, \text{else} \end{cases} \tag{6.10}$$

Similarly, we define a *lower BNT bound* by distributing the number of branches not taken equally among $p_1...p_{n-1}$. Intuitively, each query that lies with one point below this line in Figure 6.7 cannot reach the desired number of output tuples because no subsequent branch is allowed to induce more BNT than the previous one. Thus, we define the lower BNT bound as:

$$Lower\_BNT\,Bound(p) = \begin{cases} tups_{out}, \text{if p = n} \\ tups_{out}, \text{if Lower BNT Bound(p)} < tups_{out} \\ \frac{BNT_{samp} - tups_{out} - ((p-1) * tups_{in})}{n-1}, \text{else} \end{cases} \tag{6.11}$$

Using the lower and upper BNT bound, we can restrict the search space for our example query ([80, 70, 50, 10]) in Figure 6.7. Accesses to $[col_1, ..., col_4]$ have to be in the interval between the selectivity ranges $[67, 50, 10, 10]$ and $[100, 95, 66, 10]$. As shown, using the upper and lower BNT bound, we are able to restrict the search space significantly.

6.5.2 Learning Algorithm

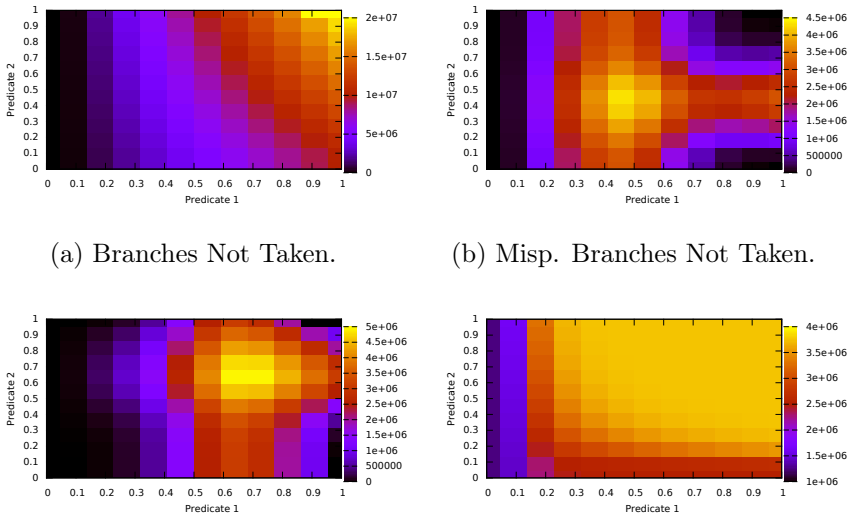
The main challenge for an algorithm that approximates selectivities of individual predicates is the ability to distinguish different queries. We showed that this distinction is possible for a query using one predicate (see Figure 6.2) and two predicates (see Figure 6.6). However, for a multi-selection query, we measure performance counters of the entire PEO execution and thus have to infer the individual predicate selectivity.

In our progressive optimization approach, we exploit four performance counters: 1) branches not taken, 2) branches taken mispredictions, 3) branches not taken mispredictions, and 4) L3 accesses. These counters can be gathered simultaneously on modern CPUs. In Figure 6.8, we plot the predictions of these counters for a selection with two predicates on 10M tuples as a 2D heat map. The predictions are calculated by our cost models presented in Section 6.4. The selectivity of the first predicate is shown on the x-axis and the selectivity of the second on the y-axis. In general, we can distinguish two queries if they differ in at least one of these counter values. In Figure 6.8, a counter value is represented by the color of the square at the intercept point of the selectivities. For example, a query using two predicates with 40% and 20% selectivity differs from a query using two predicates with 20% and 40% selectivity in the number of mispredicted branches not taken (see Figure 6.8b).

To learn the selectivities of individual predicates, we apply a non-linear optimization algorithm. We use the open-source library *NLopt* [NLo17]. This library supports different algorithms including gradient-based and derivation free algorithms. Based on an extended evaluation of all available algorithms regarding their correctness and speed, we choose the *Nelder-Mead simplex algorithm* [NM65] as a local optimization algorithm because it performs best for our selectivity estimations.

Based on this decision, we define the following minimization function for the non-linear optimization that uses the difference between the sampled value (samp) and our estimation (est) to determine the costs of a PEO in terms of cache events.

$$\begin{aligned} Costs = (BNT_{samp} - BNT_{est}) + (L3_{samp} - L3_{est}) \\ + (BRNotMP_{samp} - BRNotMP_{est}) \\ + (BRTakMP_{samp} - BRTakMP_{est}) \end{aligned} \quad (6.12)$$



(a) Branches Not Taken.

(b) Misp. Branches Not Taken.

(c) Misp. Branches Taken.

(d) L3 Accesses.

Figure 6.8: Two Predicate Prediction.

To restrict the optimization effort, we utilize the lower BNT bound (see Equation 6.11) and upper BNT bound (see Equation 6.10) as boundaries for the optimization. Additionally, we specify an absolute tolerance from the previous iteration and a maximum iteration count as termination criteria.

The algorithm proceeds as follows. In the first iteration, the algorithm calculates the minimization function from a given start point. Based on the calculated function value, the algorithm internally changes the individual selectivities to values between the upper and lower bound and recalculate the minimization function for these new values. The optimization terminates if the maximum iteration count is reached or the current optima differs less than specified by the absolute tolerance from the last iteration. In our tests, a maximum iteration count of 10K and an absolute tolerance of one result in the best estimations. As a result, the algorithms returns a selectivity estimation for each individual predicate.

6.5.3 Selection a Starting Point

In our optimization approach, our system of linear equation is under-defined because we cannot utilize as many performance counters as individual predicates exists. Furthermore, it is possible that two PEOs of the same query induce the same performance counter value in each exploited counter. In our evaluation, this scenario mostly occurs if on query induce an equal distribution of accesses and the other an extreme skew.

As a consequence, when performing the non-linear optimization only once, we could potentially terminate on a local optima. To encounter this problem, we specify a set of start points for our non-linear optimization algorithm and perform the optimization multiple times.

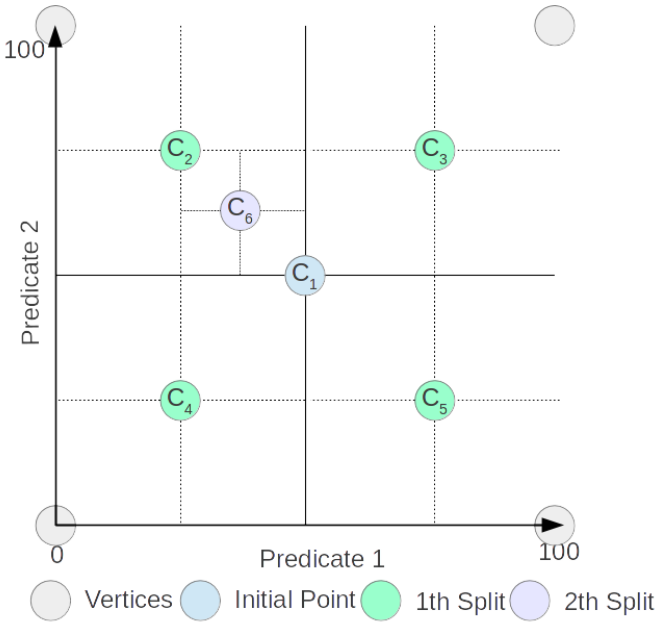


Figure 6.9: Start Point Selection.

In Figure 6.9, we outline our approach to create a set of start points for a two-dimensional search space. At first, we create start points at the vertices of each dimension. For our example query, we would create start points [0,0], [0,100], [100,100], and [100,0]. After that, we set the initial point using our null hypothesis. As our null hypothesis, we assume that the overall query selectivity distributes evenly among the predicates. Using this as a start point, we split the search space in 2^n sub-spaces. In Figure 6.9, the query induces a selectivity of 25% and thus we create the initial point C_1 which splits the search space into four equally sized squares.

For each additional start point, we search for the largest sub-space and return its centroid as a start point for the non-linear optimization algorithm. In Figure 6.9, the start points in the first splitting phase are C_2, C_3, C_4, and C_5. If an additional start point is required, C_6 would be created. Based on this algorithm, we create start points that are evenly distributed among the search space to avoid the termination on a local optima during the non-linear optimization. Additionally, we explore the largest unseen part of the search space for each new start point.

6.5.4 Progressive Optimization Algorithm

In Figure 6.10, we present our progressive optimization algorithm which drives the vectorized execution. First, we measure one vector execution and sample the performance counters specified in Section 6.5.2. As next steps, we repeatedly generate a start point (see Section 6.5.3), run the non-linear optimization algorithm (see Section 6.5.2), and compare the current optima against the previous optima. This sequence terminates if either no better local optima was found in the previous n iterations or if an overall iteration maximum m is reached. The values of n and m represent a trade-off between the quality of the estimation and the required optimization time. In our experiments, $n < 5$ and $m = 2^p$ with p as the number of predicates lead to the best trade-off between optimization time and estimation precision.

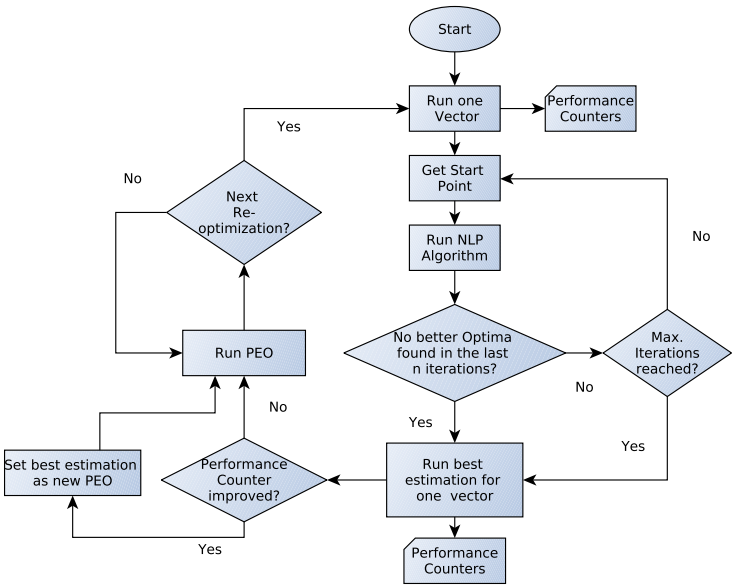


Figure 6.10: Optimization Sequence.

After the sequence terminates, we reorder the predicates according to the best estimation found so far. A JIT-compiled system like Hyper [Neu11] would compile a new binary for the new predicate evaluation order. In contrast, a vectorized system like Vectorwise [SZB11] could have pre-compile primitives that are chained in the new predicate evaluation order. Using the new PEO, we execute another vector and sample the required performance counters again. If the performance counter values improve, the PEO is used for the consecutive vectors. If they deteriorate, the old PEO is reestablished. Finally, vector execution continues until the next optimization cycle is scheduled.

6.5.5 Skew and Correlation

Skewed data distributions and correlated attributes are two of the traditional challenges of database query optimization [Bea13, Mea04, KD98, Sea01]. Both are cases in which the quality of an optimized QEP may be low because the cost estimator cannot accurately infer factors such as selectivity coefficients, the probability of collisions when building hashes, or data access locality. However, many tuning parameters that impact cost estimation such as buffer sizes, hash functions, or selection strategies can be adapted during query execution. Thus, processing in fine-grained partitions might help to remedy poor decisions based on unpredictable data characteristics such as skew and correlation. In fact, our approach effectively renders high quality decisions at query compilation time unnecessary because it provides better and more adaptive information at run-time.

In our approach, skew is implicitly detected by periodically inspecting the performance of the execution. Thus, if the value distribution of the data set changes for a subset, we could detect this during the next optimization run and change the plan accordingly. In contrast to skew which affects a single attribute, correlation affects a combination of attributes and violates the underlying assumption of modern query optimizers that the value distribution of a seen data subset applies to the entire data set. It introduces low quality estimates because not seen data subsets may exhibit another distribution. As a consequence, selectivities might change significantly. We handle correlation in our approach by periodically execute different PEOs. With an increasing number of optimization runs per execution, more PEOs are executed and thus the probability that a better PEO is missed cause of correlations is reduced. Furthermore, by determining the amount of data seen by recent PEOs, we are able to introduce special PEO changes to explore unseen data subsets and thus detect correlations.

6.6 Evaluation

In this section, we evaluate our progressive optimization approach for different selectivity and value distributions based on the TPC-H benchmark. First, we present our experimental setup in Section 6.6.1. Then, we start our evaluation by comparing the execution of Q6 with and without our progressive optimization in Section 6.6.2. After that, we evaluate different selectivity distributions in Section 6.6.3 and different value distributions in Section 6.6.4. In Section 6.6.5 and Section 6.6.6, we showcase how sortedness can be exploited to reorder QEPs involving join operators. Finally, we investigate the overhead of our progressive optimization approach in Section 6.6.7.

6.6.1 Experimental Setup

For our evaluation, we implement the original TPC-H query six (Q6) and several modifications in a C++ prototype. We utilize the common data generator to create a data set using scaling factor 100. This translates into approximately 4,7 GB of data per column of the lineitem table and approximately 600M tuples.

We evaluate our prototype on a Intel Xeon E5-2630 v2 processor. It contains six physical cores at 2.6 GHz frequency and provides 12 logical cores using hyper threading. Additionally, each core utilizes a separate 32 KB L1 cache for data and instructions and a unified 256 KB L2. All cores share a 15 MB L3 cache.

6.6.2 TPC-H Common Case

Figure 6.11 shows the execution of all predicate evaluation orders including the best and the worst for the five predicates in Q6 (120 possible orders). Q6 of the TPC-H benchmark suite is defined as follows:

```
select sum(l_extendedprice * l_discount) as revenue
from lineitem
where l_shipdate >= date '1994-01-01'
  and l_shipdate < date '1995-01-01'
  and l_discount between 0.06 - 0.01 and 0.06 + 0.01
  and l_quantity < 24
```

Figure 6.11 includes the slowest PEO with predicates ordered in descending selectivity order and the fastest PEO with predicates ordered in ascending selectivity order. The black line represents the base line for our evaluation which executes Q6 without progressive optimization. Therefore, we choose one PEO and stick to it for all vectors. The green line represent the run-time with progressive optimization and the same PEO as the non-optimized execution as the start PEO. Overall, this query executes 600 vectors with 1M tuples per vector. We start our optimization approach (see Section 6.5.4) after each 10th vector. The results are sorted on total run-time of the common execution pattern without progressive optimization.

As shown in Figure 6.11, our approach improves run-time for this query regardless of the first initial PEO choice. However, the actual improvement fluctuates to some degree based on the start PEO and thus the time necessary to converge to the best PEO. Our approach improves execution time because we converge to the fastest PEO and react to changing selectivities and data properties during execution.

Our optimization approach improves performance more than the fastest PEO because we react to changes in selectivities during run-time. In particular, Q6 contains a range query which selects tuples between 1994 and 1995. The resulting code implements one predicate that evaluates `year > 1994`

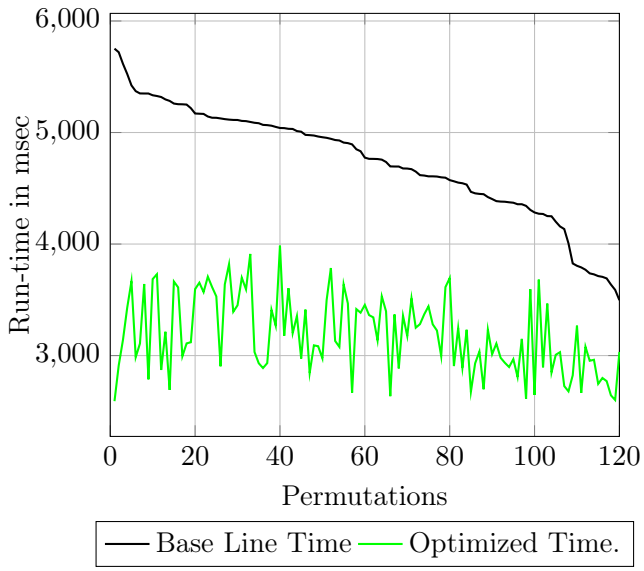


Figure 6.11: TPC-H Common Case.

and one predicate `year < 1995`. In general, there are three different PEOs which are best for a given year range. First, tuples with a year value before 1994 should be evaluated by the `year > 1994` first because all of them are will disqualify. Second, tuples with a year value later than 1995 should be evaluated by the `year < 1995` first. Third, tuples in between 1994 and 1995 should be evaluated by other predicates first because they qualify both year predicates. Therefore, the query contains at least three best PEOs. In contrast to running the fastest PEO for all vectors, our approach change PEOs during run-time based on sampling values.

6.6.3 Selectivity Distribution

In this experiment, we explore the impact of selectivity on our progressive optimization approach. We execute Q6 using different selectivities on ship date and show the results in Figure 6.12. For each selectivity, we plot the minimum, maximum, and average run-time of the common execution pattern without progressive optimization (base line). Additionally, we plot the average run-time using progressive optimization and a re-optimization interval of 10, 75, or 200 vectors.

For a selectivity below 0.1%, the average execution time using our progressive optimization approach differs up to a factor of two from the minimal base line execution time. In this selectivity range, the position of the ship date predicate is vital for the resulting query performance. Because the impact is so huge, the necessary optimization time transfer directly to a sub-optimal run-time. Our approach converges slowly to the best PEO if we

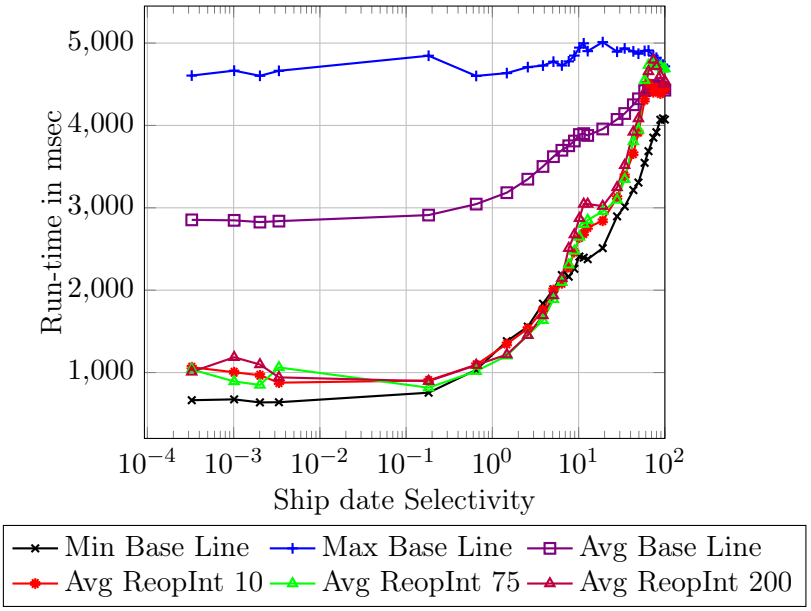


Figure 6.12: Q6 with varying Ship date Selectivity.

start our optimization using a PEO which evaluates the ship date predicate in the middle of the PEO. In this case, our optimization algorithm requires multiple steps to converge to the optimal PEO. In contrast, if the ship date predicate is evaluated early or late in the PEO, our progressive optimization algorithm converges very fast to the optimal PEO.

In the selectivity range between 0.1% and 10%, the average run-time using progressive optimization reaches the minimal base line run-time. Thus, our optimization algorithm performs very efficiently in this selectivity range. For selectivities over 10%, our optimization algorithm slightly differs from the minimal base line run-time with the largest difference for very high selectivities. In general, large selectivities are hard to detect by our algorithm because the high number of branches not taken leads to a high number of possible selectivity distributions.

Overall, progressive optimization improves run-time up to a factor of three compared to the average run-time and up to a factor of 4.5 compared to the worst case run-time. Thus, we efficiently alleviate bad initial PEOs and make the overall query execute more robust.

6.6.4 Sortedness

In this experiment, we explore the impact of sortedness on progressive optimization. We examine the run-time of Q6 on differently sorted data sets and present the results in Figure 6.13. In Figure 6.13a, the data set is sorted on the ship date column in ascending order. In general, shorter re-optimization

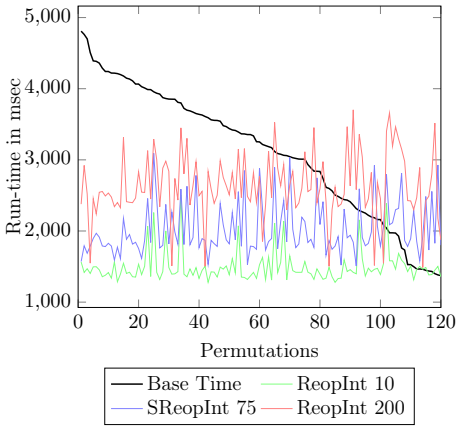
intervals result in better run-times for sorted data. The main reason for that is the point in time, at which the optimization algorithm detects that a better PEO exists and change to it. For Q6 with a lower and upper bound on the ship date column, there are three different optimal PEOs during execution. In the first data partition, it is beneficial to evaluate the lower ship date bound (> 1994) first because it has an effective selectivity of 0%. In the middle partition of the data set where ship dates fall in between the lower and upper ship date bound, both ship date predicates should be evaluated as late as possible in the PEO. Finally, in the last partition, the upper ship date bound (< 1995) should be evaluated first to eliminate unnecessary overhead.

Based on these partitions, the optimal PEO changes during query execution. The larger the re-optimization interval, the later a transition between partitions will be detected. In the worst case, a transition is bypassed and an entire partition is executed using a sub-optimal PEO. This situation occurs for larger optimization intervals of 75 and 200 vectors and lead to increased run-times. Finally, for faster initial PEOs (Permutation 80-120), this translates to slower run-times for progressive optimization compared to the common execution pattern. However, using progressive optimization and a re-optimization interval of ten, we still introduce robust query execution with run-times faster or at least as far as the common execution pattern.

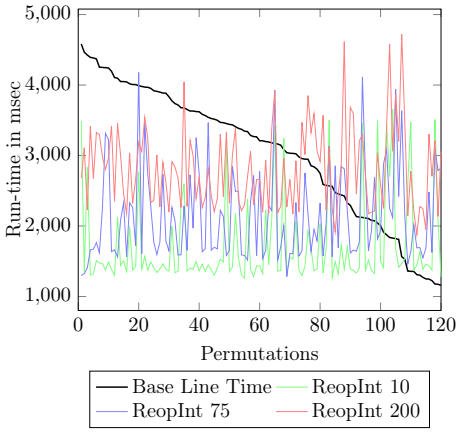
In Figure 6.13c, the data set is randomly distributed. As a result, each predicate has an arbitrarily selectivity on each vector. Therefore, the underlying assumption of progressive optimization that we can predict future run-time based on sampling current vector execution, is no longer valid. The re-optimization interval of 10 leads to the best run-times because it reacts most rapidly to changing value distributions. However, compared to the sorted data set, the run-times are increased for faster initial PEOs (Permutations 90-120). With larger optimization intervals, the improvements of progressive optimization decrease further. Using a re-optimization interval of 200, the run-time is almost always above base line execution.

In Figure 6.13b, we use Knuth's shuffling algorithm [Knu73] to redistribute the ship date column. To introduced a clustered data set, we shuffle lineitems based on the ship date column within the time frame of a month. This represents a middle ground between a sorted and random data set. Compared to a sorted data set, run-times increase slightly for small re-optimization intervals and moderate for large re-optimization intervals. Compared to a random data set, the overall run-time is still improved.

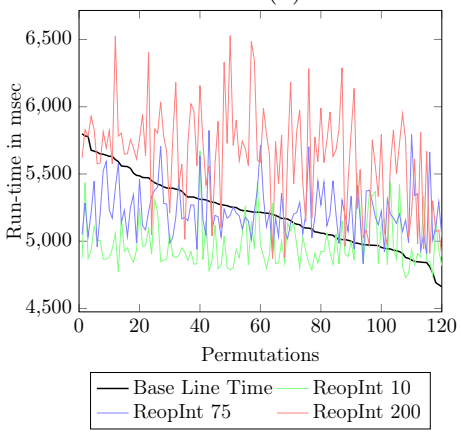
Overall, the improvements of progressive optimization decrease for randomly distributed data sets. In particular, a decreased number of initial PEOs are improved. In general, a short re-optimization interval leads to the best results. However, in the next section, we present a method to detect sortedness of a data set which could be exploited to decide if our approach should be applied and which optimization interval should be used.



(a) TPCH Sorted Data Set.



(b) TPCH Clustered Data Set.



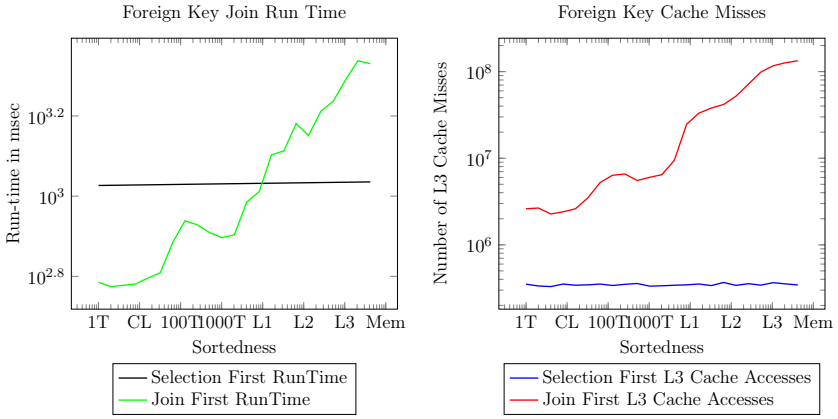
(c) TPCH Random Data Set.

Figure 6.13: Q6 on Different Value Distributions.

6.6.5 Sortedness and Expensive Predicates

In the previous section, we showed how important sortedness is if we try to choose the optimal PEO. In this experiment, we utilize performance counters to detect the sortedness of a data set. In Figure 6.14, we plot run-time and cache misses for a query using an expensive selection and a foreign key join. On the x-axis, we show different degrees of sortedness using Knuth shuffle ranging from a sorted data set (1T) to a random data set (Mem). In between, the shuffle distance hit the size of a cache line (CL), L1, L2, or L3 cache. On these data sets, we run a query that either executes a selection or a foreign key join first and the other operator afterwards.

As shown in Figure 6.14a, there is a break even point for run-time. This point is reached if the shuffle distance exceeds twice the L1 cache size. For a sortedness below this point, it is cheaper to perform the join before applying



(a) Run-time. (b) Cache Misses.

Figure 6.14: Foreign Key Join.

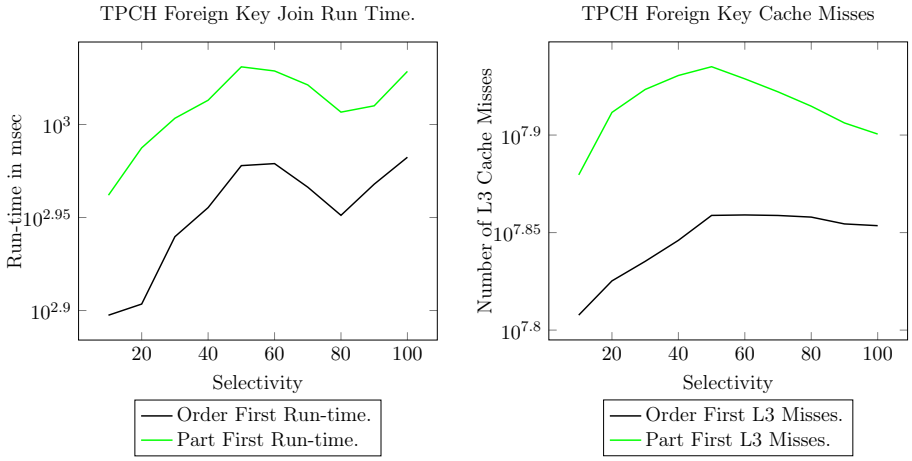
the selection. The join is so cheap because such a sortedness introduces a highly local access pattern which induces only few cache misses. In contrast, if the sortedness spreads over this point and thus the access pattern performs more random memory accesses, the join becomes more expensive and thus the selection should be applied first.

Note that, this kind of sortedness analysis can only be derived from performance counter. In particular, counting the number of qualifying tuples per vector is not sufficient. Therefore, measuring the number of cache misses (see Figure 6.14b) allows us to infer the sortedness. In this scenario, the trend of the run-time and the number of cache misses correlate. Thus, we could derive sortedness and reorder the operations using our progressive optimization approach.

6.6.6 Sortedness for Foreign Key Join

In this experiment, we use our approach to optimize the join order in a QEP. In Figure 6.15, we join the lineitems table of the TPC-H benchmark with the order and part table in two different orders. On the x-axis, we show the selectivity of both joins. Commonly, a query optimizer would join lineitems first with the part table because it is about eight times smaller than orders table. However, as shown in Figure 6.15a, joining orders first leads to an improved run-time for all selectivities. The main reason for this is, that lineitems and orders are co-clustered. In contrast, the access pattern to the part table is random. This co-clusteredness leads to an improved accesses pattern with less cache misses as shown in Figure 6.15b.

In our approach, we exploit Equation 6.2 from Section 6.4.1 to determine if a join is executed on a co-clustered table pair. Using Equation 6.2, we estimate the expected number of cache misses for a random access pattern.



(a) Run-time. (b) Cache Misses.

Figure 6.15: Foreign Key Join with different Orders.

Then, we compare these values against the sampled cache misses. If they match, we might reorder the join order; thus, eventually switching to a join order where a co-clustered join is executed first. In contrast, when we sample much less cache misses than expected, we gain knowledge that the we probably execute a co-clustered join first and do not have to reorder. It is important to note, that this kind of join order optimization can be exploited in our approach in addition to the branch not taken/cache miss sampling approach which we use for multi-level selection queries.

6.6.7 Overhead

In this experiment, we evaluate the overhead of progressive optimization using performance counters against a counter-based approach, called *enumerator-based* approach in the following. An enumerator-based approach would insert explicit counter variables into the source code after each predicate evaluation to obtain the individual selectivities. In contrast, we use non-invasive performance counters to approximate these selectivities.

The overhead of progressive optimization is comprised of two components. First, we have to compare the exploitation of performance counters against the usage of explicit counter variables. In Figure 6.16, we measure overhead for both variants for different predicate counts. As shown, for larger predicate counts, the enumerator-based approach almost doubles query run-time. In contrast, performance counter do not impact run-time. This observation follows Intel's statement that performance counters do not or only minimally impact the execution performance [Int12b]. Thus, during optimization cycles, the enumerator-based approach would double run-time. In contrast, performance counter introduce virtually no costs.

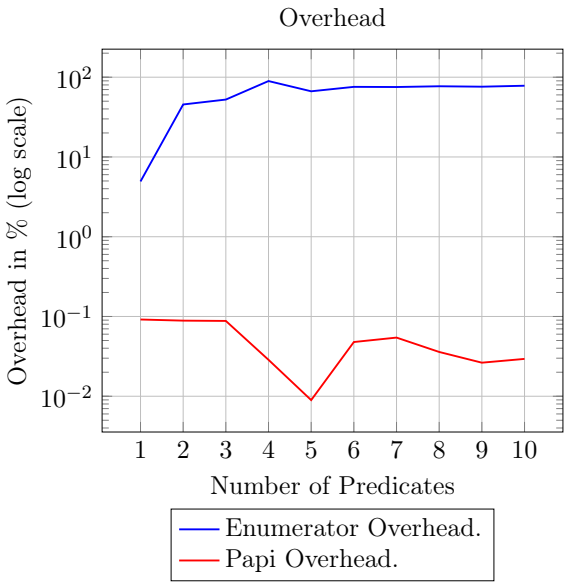


Figure 6.16: Overhead.

Second, we have to compare the algorithm to infer the individual predicate selectivity used by our approach against a similar algorithm for the enumerator-based approach. In our evaluation we showed, that optimization overhead contribute only minor to total run-time. We assume, a comparable algorithm for an enumerator-based approach should perform similar. Finally, progressive optimization using performance counter rely solely on existing implementations. In contrast, an enumerator-based approach has to maintain implementations with and without counter variables for each operator. Furthermore, an enumerator-based approach has to modify existing code which is not always possible.

6.7 Conclusion and Future Work

This chapter provides the necessary cost models to enable performance counters for progressive optimization. Our progressive optimization approach using performance counters avoids worst case predicate evaluation orders efficiently. Using progressive optimization, we improve run-time up to a factor of three compared to average run-time and up to a factor of 4,5 compared to worst case run-time. Thus, we efficiently alleviate slow initial PEOs and make the overall query execute more robust. At the same time, the optimization overhead could be restricted by fine tuning the termination criteria of the underlying non-linear optimization algorithm, the number of optimizations during execution, and the effort that is spent to find the best optimization result.

Our evaluation showed, that expect for a random data distribution, we almost always improve run-time compared to the common execution pattern through periodically re-optimizing sub-optimal PEOs. Finally, we showed that the impact of sortedness, skew, and correlation can be alleviated by our approach.

Future work based on our approach should integrate other relational operators into our optimization approach. Additionally, if new performance counters become available through new processor technologies, they should be implemented to improve estimations.

Chapter 7

Conclusion

7.1 Summary

In this thesis, we addressed different challenges for database systems running on modern CPUs. We focused on the area of query execution and presented four approaches:

1. We enable SIMD instructions for tree traversal to significantly speed-up tree operations.

2. We propose an unified model to optimize parallel query execution based on characteristics of modern CPUs.

3. We measure and analyze different workflows on modern CPUs using performance counters to create an in-dept knowledge of how queries exploit the capabilities of modern CPUs.

4. We propose a non-invasive progressive optimization approach which optimizes query execution during run-time based on CPU and query characteristics.

The contributions of this thesis significantly improve the execution of queries on modern CPUs. Our results point out, that hardware-conscious query execution will largely contribute to the performance of future database systems. With an ever-growing diversity of different hardware architectures and features, a hardware-conscious approach could adaptively optimize query execution. To enable a hardware-conscious approach, we lay the foundations by analyzing different operators and proposing algorithms to exploit this knowledge. With this thesis, we advance the research field of query execution on modern CPUs significantly.

7.2 Future Work

Future work might focus on the following areas:

1. The the impact of multi-threading, multi-core, and many-core architectures on different aspects of our tree adaptations should be further investigated. Especially, the impact of SIMD instructions on concurrently used index structures and the execution on GPUs could be interesting.

2. The exploitation of work sharing opportunities between queries might be an interesting extension to our QTM model. Furthermore, a cost model that predicts the costs of different task configurations on different hardware architectures could significantly improve the query optimization and execution using QTM.

3. The in-depth analysis of the relational selection operator should be extended to other relational operators. The results might be used to improve query optimizer in modern DBMS. Furthermore, the results should be integrated into our non-invasive progressive optimization approach.

4. If new performance counters become available through new processor technologies, they might be exploited to predict the behavior of operators more precisely.

The overall goal of future work should be the extension of our non-invasive progressive optimization approach such that our approach become capable to optimize entire query execution plans with different operators. In the best case, all relational operators could be integrated into our approach. An optimizer using our progressive optimization approach would be able to react to different query and hardware characteristics during run-time and thus enables robust and near-optimal query execution.

Bibliography

[AAA13] Ioannis Alagiannis, Manos Athanassoulis, and Anastasia Ailamaki. Scaling up analytical queries with column-stores. In *DBTest*, 2013.

[Aea99] Anastassia Ailamaki et al. Dbmss on a modern processor: Where does time go? In *VLDB*, 1999.

[Aea09] Dan Alcantara et al. Real-time parallel hashing on the GPU. In *ACM Trans. Graph.*, volume 28, 2009.

[AMD13] AMD. *AMD64 Architecture Programmer's Manual*. http://developer.amd.com/resources/developer-guides-manuals/, 2013.

[AMH08] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: How different are they really? In *SIGMOD*, 2008.

[AV88] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. In *Commun. ACM*, 1988.

[AVX08] *Intel Advanced Vector Extensions Programming Reference*. http://software.intel.com/en-us/avx/, 2008.

[BBD05] Shivnath Babu, Pedro Bizarro, and David DeWitt. Proactive re-optimization. In *SIGMOD*, 2005.

[BDFC00] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious b-trees. In *FOCS*, 2000.

[Bea83] Dina Bitton et al. Parallel algorithms for the execution of relational database operations. In *TODS*, 1983.

[Bea96] Luc Bouganim et al. Dynamic load balancing in hierarchical parallel database systems. In *VLDB*, 1996.

[Bea99] Peter Boncz et al. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, 1999.

[Bea05] Peter A. Boncz et al. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, 2005.

[Bea09] He Bingsheng et al. Relational query coprocessing on graphics processors. In *ACM Trans. Database Syst.*, 2009.

[Bea11a] Spyros Blanas et al. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *SIGMOD*, 2011.

[Bea11b] Matthias Boehm et al. Efficient in-memory indexing with generalized prefix trees. In *BTW*, 2011.

[Bea13] Cagri Balkesen et al. Main-memory hash joins on multi-core cpus : Tuning to the underlying hardware. In *ICDE*, 2013.

[BFCK06] Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. Cache-oblivious string b-trees. In *PODS*, 2006.

[BGB98] Luiz André Barroso, Kourosh Gharachorloo, and Edouard Bugnion. Memory system characterization of commercial workloads. In *ISCA*, 1998.

[BLP11] Spyros Blanas, Yinan Li, and Jignesh M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *SIGMOD*, 2011.

[BM70] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *SIGFIDET*, 1970.

[Bro15] David Broneske, et al. Database scan variants on modern cpus: A performance study. In *IMDM*, 2015.

[BU77] Rudolf Bayer and Karl Unterauer. Prefix b-trees. In *ACM Trans. Database Syst.*, 1977.

[Cea08] Jatin Chhugani et al. Efficient implementation of sorting on multi-core simd cpu architecture. In *VLDB*, 2008.

[Cea09] John Cieslewicz et al. Cache-conscious buffering for database operators with state. In *DaMoN*, 2009.

[CGM01] Shimin Chen, Phillip B. Gibbons, and Todd C. Mowry. Improving index performance through prefetching. In *SIGMOD*, 2001.

[CK85] George P. Copeland and Setrag N. Khoshafian. A decomposition storage model. In *SIGMOD*, 1985.

[Com79] Douglas Comer. Ubiquitous b-tree. In *ACM Comp. Surv.*, 1979.

[CR08] John Cieslewicz and Kenneth A. Ross. Data partitioning on chip multiprocessors. In *DaMoN*, 2008.

[CRG07] John Cieslewicz, Kenneth A. Ross, and Ioannis Giannakakis. Parallel buffers for chip multiprocessors. In *DaMoN*, 2007.

[DH14] Anshuman Dutt and Jayant R. Haritsa. Plan bouquets: Query processing without selectivity estimation. SIGMOD, 2014.

[Eea96] Richard J. Eickemeyer et al. Evaluation of multithreaded uniprocessors for commercial application environments. In *ISCA*, 1996.

[Fly72] M. J. Flynn. Some computer organizations and their effectiveness. In *IEEE Transactions on Computers*, 1972.

[Gea04] Naga K. Govindaraju et al. Fast computation of database operations using graphics processors. In *SIGMOD*, 2004.

[Gea06] Naga K. Govindaraju et al. Gputerasort: high performance graphics co-processor sorting for large database management. In *SIGMOD*, 2006.

[GI96] Minos N. Garofalakis and Yannis E. Ioannidis. Multi-dimensional resource scheduling for parallel queries. In *SIGMOD*, 1996.

[GL01] Goetz Graefe and P. Larson. B-tree indexes and cpu caches. In *ICDE*, 2001.

[GR14] M. B. Giles and I. Reguly. Trends in high-performance computing for engineering calculations. In *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 2014.

[Gra90] Goetz Graefe. Encapsulation of parallelism in the volcano query processing system. In *SIGMOD*, 1990.

[Gra94] Goetz Graefe. Volcano an extensible and parallel query evaluation system. In *IEEE Trans. on Knowl. and Data Eng.*, 1994.

[HA03] Stavros Harizopoulos and Anastassia Ailamaki. A case for staged database systems. In *CIDR*, 2003.

[HA04] Stavros Harizopoulos and Anastassia Ailamaki. Steps towards cache-resident transaction processing. In *VLDB*, 2004.

[HA05] Stavros Harizopoulos and Anastassia Ailamaki. Stageddb: Designing database servers for modern hardware. In *IEEE Data Eng. Bull.*, 2005.

[Hea] Max Heimel et al. Hardware-oblivious parallelism for in-memory column-stores. In *PVLDB*, volume 6.

[Hea06] Stavros Harizopoulos et al. Performance tradeoffs in read-optimized databases. In *VLDB*, 2006.

[Hea07a] Wook-shin Han et al. Progressive optimization in a shared-nothing parallel database. In *SIGMOD*, 2007.

[Hea07b] Nikos Hardavellas et al. An analysis of database system performance on chip multiprocessors. In *ISCA*, 2007.

[Hea07c] Nikos Hardavellas et al. Database servers on chip multiprocessors: Limitations and opportunities. In *CIDR*, 2007.

[Hon92] Wei Hong. Exploiting inter-operation parallelism in xprs. In *SIGMOD*, 1992.

[HP11] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 5th edition, 2011.

[HS89] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. In *IEEE Trans. Comput.*, 1989.

[HSA05] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. Qpipe: A simultaneously pipelined relational query engine. In *SIGMOD*, 2005.

[HZW02] Steffen Heinz, Justin Zobel, and Hugh E. Williams. Burst tries: a fast, efficient data structure for string keys. In *ACM Trans. Inf. Syst.* ACM, 2002.

[Iea07] Hiroshi Inoue et al. Aa-sort: A new parallel sorting algorithm for multi-core simd processors. In *PACT*, 2007.

[Int12a] Intel. *Intel® 64 and IA-32 Architectures Optimization Manual*. 2012.

[Int12b] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. https://software.intel.com/en-us/articles/intel-sdm, 2012.

[Int17a] Intel. https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors, 2017. Accessed Mai 2017.

[Int17b] Intel. https://software.intel.com/en-us/intel-parallel-studio-xe/details, 2017. Accessed Mai 2017.

[Jea07] Ryan Johnson et al. To share or not to share? In *VLDB*, 2007.

[JW89] N. P. Jouppi and D. W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *ASPLOS*, 1989.

[KD98] Navin Kabra and David J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD*, 1998.

[Kea98] Kimberly Keeton et al. Performance characterization of a quad pentium pro smp using oltp workloads. In *ISCA*, 1998.

[Kea06] Holger Kache et al. Pop / fed : Progressive query optimization for federated queries in db2. In *EDBT*, 2006.

[Kea10] Changkyu Kim et al. Fast: fast architecture sensitive tree search on modern cpus and gpus. In *SIGMOD*, 2010.

[Kea11] Changkyu Kim et al. Designing fast architecture-sensitive tree search on modern multicore/many-core processors. In *ACM Trans. Database Syst.*, 2011.

[Kea12] Tim Kaldewey et al. Gpu join processing revisited. In *DaMoN*, 2012.

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Vol. III: Sorting and Searching.* Addison-Wesley, 1973.

[Lea98] Jack L. Lo et al. An analysis of database workload performance on simultaneous multithreaded processors. In *ISCA*, 1998.

[Lea12] Susana Ladra et al. Exploiting simd instructions in current processors to improve classical string algorithms. In *ADBIS*, 2012.

[Lea14] Viktor Leis et al. Morsel-driven parallelism : A numa-aware query evaluation framework for the many-core age. In *SIGMOD*, 2014.

[LKN13] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *ICDE*, 2013.

[LLS11] Justin J. Levandoski, David B Lomet, and Sudipta Sengupta. The bw-tree : A b-tree for new hardware platforms. In *IEEE*, 2011.

[LP13] Yinan Li and Jignesh M. Patel. Bitweaving. In *SIGMOD*, 2013.

[LR05] Bin Liu and Elke A. Rundensteiner. Revisiting pipelined parallelism in multi-join query processing. In *VLDB*, 2005.

[LT92] Hongjun Lu and Kian-Lee Tan. Dynamic and load-balanced task-oriented datbase query processing in parallel systems. In *EDBT*, 1992.

[Man02] Stefan Manegold, et al. Generic database cost models for hierarchical memory systems. In *VLDB*, 2002.

[MBK02] Stefan Manegold, Peter Boncz, and Martin Kersten. Optimizing main-memory join on modern hardware. In *IEEE Trans. on Knowl. and Data Eng.*, 2002.

[MDO94] Ann Marie Grizzaffi Maynard, Colette M. Donnelly, and Bret R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *ASPLOS*, 1994.

[Mea04] Volker Markl et al. Robust query processing through progressive optimization. In *SIGMOD*, 2004.

[Mic17] Microsoft. http://msdn.microsoft.com/en-us/library/, 2017. Accessed Mai 2017.

[MOW97] Stefan Manegold, Johann K. Obermaier, and Florian Waas. Load balanced query evaluation in shared-everything environments. In *Euro-Par*, 1997.

[MSM04] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for Parallel Programming.* Addison-Wesley Professional, first edition, 2004.

[Neu11] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. In *VLDB*, 2011.

[NLo17] NLopt. http://ab-initio.mit.edu/wiki/index.php/nlopt, 2017. Accessed Mai 2017.

[NM65] J. A. Nelder and R. Mead. A simplex method for function minimization. In *The Computer booktitle*, 1965.

[PAA13] Iraklis Psaroudakis, Manos Athanassoulis, and Anastasia Ailamaki. Sharing data and work across concurrent analytical queries. In *VLDB*, 2013.

[PAP17] PAPI. http://icl.cs.utk.edu/papi/, 2017. Accessed Mai 2017.

[Pat15] Jason Robert Carey Patterson. http://www.lighterra.com/papers/modernmicroprocessors/, 2015. Accessed Mai 2017.

[Pea90] Hamid Pirahesh et al. Parallelism in relational data base systems: Architectural issues and design approaches. In *DPDS*, 1990.

[Pea01] Sriram Padmanabhan et al. Block oriented processing of relational database operations in modern computer architectures. In *ICDE*, 2001.

[Pea13] Iraklis Psaroudakis et al. Task scheduling for highly concurrent analytical and transactional main-memory workloads. In *ADMS*, 2013.

[Pir13] Holger Pirk, et al. Cpu and cache efficient management of memory-resident databases. ICDE, 2013.

[PMJA01] S. Padmanabhan, T. Malkemus, a. Jhingran, and R. Agarwal. Block oriented processing of relational database operations in modern computer architectures. In *IEEE Comput. Soc*, 2001.

[Pos17] PostgreSQL. http://www.postgresql.org/, 2017. Accessed Mai 2017.

[PRR15] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. Rethinking simd vectorization for in-memory databases. In *SIGMOD*, 2015.

[RBH+95] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. The impact of architectural trends on operating system performance. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, 1995.

[RBZ13] Bogdan Răducanu, et al., Peter Boncz, and Marcin Zukowski. Micro adaptivity in vectorwise. In *SIGMOD*, 2013.

[Rea98] Parthasarathy Ranganathan et al. Performance of database workloads on shared-memory systems with out-of-order processors. In *ASPLOS*, 1998.

[Rea13] Vijayshankar Raman et al. Db2 with blu acceleration: So much more than just a column store. In *VLDB*, 2013.

[Ros02] Kenneth A. Ross. Conjunctive selection conditions in main memory. In *PODS*, 2002.

[Ros04] Kenneth A Ross. Selection conditions in main memory. In *TODS*, 2004.

[Ros07] K.A. Ross. Efficient hash probes on modern processors. In *ICDE*, 2007.

[RR99] Jun Rao and Kenneth A. Ross. Cache conscious indexing for decision-support in main memory. In *VLDB*, 1999.

[RR00] Jun Rao and Kenneth A. Ross. Making b+- trees cache conscious in main memory. In *SIGMOD*, 2000.

[Sea01] Michael Stillger et al. Leo - db2 ' s learning optimizer. In *VLDB*, 2001.

[Sea05] Mike Stonebraker et al. C-store: A column-oriented dbms. In *VLDB*, 2005.

[Sea07] Michael Stonebraker et al. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, 2007.

[SGL09] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. k-ary search on modern processors. In *DaMoN*, 2009.

[SGL10] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. Fast integer compression using simd instructions. In *DaMoN*, 2010.

[SKN94] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. Cache conscious algorithms for relational query processing. In *VLDB*, 1994.

[Smi82] Alan Jay Smith. Cache memories. In *ACM Comput. Surv.*, 1982.

[SS00] Nathan Slingerland and Alan Jay Smith. Multimedia extensions for general purpose microprocessors: a survey. Technical Report UCB/CSD-00-1124, EECS Department, University of California, Berkeley, Dec 2000.

[SWL11] Benjamin Schlegel, Thomas Willhalm, and Wolfgang Lehner. Fast sorted-set intersection using simd instructions. In *ADMS*, 2011.

[SYT93] Eugene J. Shekita, Honesty C. Young, and Kian-Lee Tan. Multi-join optimization for symmetric multiprocessors. In *VLDB*, 1993.

[SZB11] Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. Vectorization vs. compilation in query execution. In *DaMoN*, 2011.

[TDF90] George Taylor, Peter Davies, and Michael Farmwald. The tlb-slice low-cost high-speed address translation mechanism. In *ISCA*, 1990.

[Tea97] Pedro Trancoso et al. The memory performance of dss commercial workloads in shared-memory multiprocessors. In *HPCA*, 1997.

[Tea16] Xinmin Tian et al. Effective simd vectorization for intel xeon phi coprocessors. In *Sci. Program.*, 2016.

[TGA13] Pinar Tözün, Brian Gold, and Anastasia Ailamaki. Oltp in wonderland: Where do cache misses come from in major oltp components? In *DaMoN*, 2013.

[TS90] Shreekant S. Thakkar and Mark Sweiger. Performance of an oltp application on symmetry multiprocessor system. In *ISCA*, 1990.

[Wag73] R. E. Wagner. Indexing design considerations. In *IBM Systems Journal*, 1973.

[Wil09] Thomas Willhalm, et al. Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units. In *VLDB*, 2009.

[WS94] Sholom M. Weiss and James E. Smith. *Power and power PC - principles, architecture, implementation.* Morgan Kaufmann, 1994.

[Yea12] Takeshi Yamamuro et al. Vast-tree: a vector-advanced and compressed structure for massive data tree traversal. In *EDBT*, 2012.

[ZCRS05] Jingren Zhou, John Cieslewicz, Kenneth A. Ross, and Mihir Shah. Improving database performance on simultaneous multithreading processors. In *VLDB*, 2005.

[Zea93] Mikal Ziane et al. Parallel query processing in dbs3. In *PDIS*, 1993.

[Zea08] Marcin Zukowski et al. Dsm vs. nsm: Cpu performance tradeoffs in block-oriented query processing. In *DaMoN*, 2008.

[ZF14] Steffen Zeuch and Johann-Christoph Freytag. QTM: modelling query execution with tasks. In *ADMS*, 2014.

[ZF15] Steffen Zeuch and Johann-christoph Freytag. Selection on modern cpus. In *IMDM*, 2015.

[ZFH14] Steffen Zeuch, Johann-Christoph Freytag, and Frank Huber. Adapting tree structures for processing with SIMD instructions. In *EDBT*, 2014.

[ZPF16] Steffen Zeuch, Holger Pirk, and Johann-Christoph Freytag. Non-invasive progressive optimization for in-memory databases. In *PVLDB*, volume 9, 2016.

[ZR02] Jingren Zhou and Kenneth A. Ross. Implementing database operations using simd instructions. In *SIGMOD*, 2002.

[ZR03] Jingren Zhou and Kenneth A. Ross. Buffering accesses to memory-resident index structures. In *VLDB*, 2003.

[ZR04] Jingren Zhou and Kenneth A. Ross. Buffering databse operations for enhanced instruction cache performance. In *SIGMOD*, 2004.

Selbständigkeitserklärung

Hiermit erkläre ich, Steffen Zeuch, dass

- ich diese Dissertation mit dem Titel: “Query Execution on Modern CPUs” selbständig und nur unter Verwendung der von mir gemäß § 6 Abs. 4 der Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät II, veröffentlicht im Amtlichen Mitteilungsblatt der Humboldt-Universität zu Berlin Nr. 34/2006 am 03.8.2006, angegebenen Hilfsmittel angefertigt habe;
- ich mich nicht bereits anderwärts um einen Doktorgrad beworben habe oder einen solchen besitze, und
- mir die Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät II der Humboldt-Universität zu Berlin bekannt ist, gemäß amtliches Mitteilungsblatt
  Nr. 34/2006.

Berlin, den 30.6.2017